

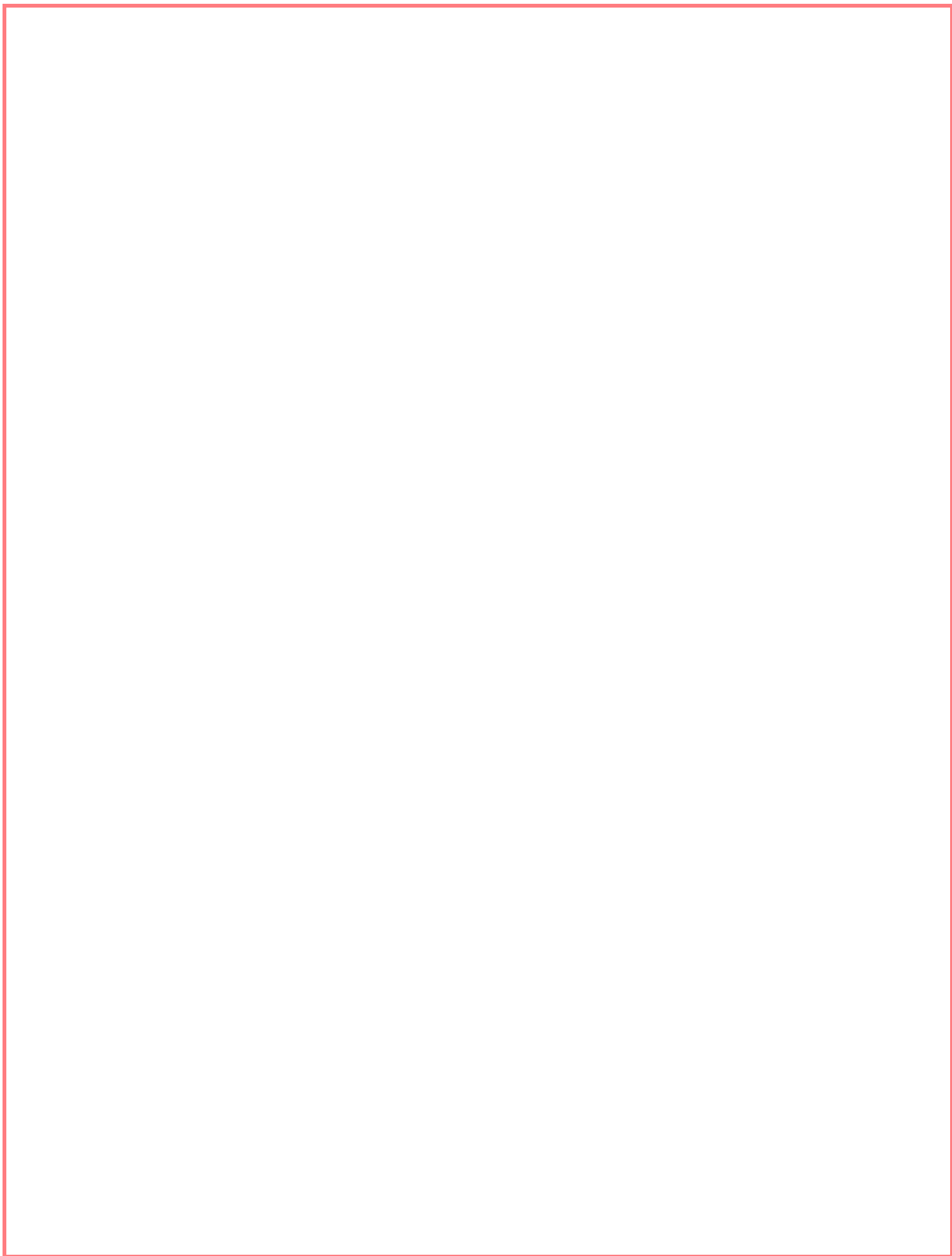


Apostila de Arduino Básico

ELABORADA PELO BOLSISTA FAPERJ

HUGO CARVALHAL SILVA

NOVEMBRO DE 2019



Índice de Figuras

Figura 1: Braços robóticos	8
Figura 2: Sensores - pHmetro.....	9
Figura 3: Sensores - sensor de gás.	9
Figura 4: Carros de controle remoto.....	10
Figura 5: Painéis solares.....	10
Figura 6: Automação Residencial.	11
Figura 7: LED.....	12
Figura 8: Push Button.....	12
Figura 9: LDR.	13
Figura 10: Potenciômetro.	13
Figura 11: Cabo Jumper.....	14
Figura 12: Protoboard.....	14
Figura 13: Resistores.	15
Figura 14: Cristal.....	15
Figura 15: Capacitor.	16
Figura 16: Transistor.....	16
Figura 17: Servo Motor.	17
Figura 18: Sensor de aproximação.	17
Figura 19: Circuito elétrico em série com três resistores.	36
Figura 20: Circuito elétrico em paralelo com três resistores.....	37
Figura 21: Resistores.	39
Figura 22: Código e padrão de cores dos resistores.	39
Figura 23: Protoboard.....	41
Figura 24: Esquema de circuito com Arduino, contendo um resistor e um LED.....	42
Figura 25: Arduino UNO, com alguns de seus componentes enumerados e explicados.	43
Figura 26: Interface do Arduino IDE.	44
Figura 27: Selecionando a porta em que o Arduino está conectado, no Arduino IDE.	46
Figura 28: Sketch que acende um LED conectado ao pino 3 do Arduino.....	47
Figura 29: Sketch que faz um LED piscar de três em três segundos.	48
Figura 30: Sketch que declara uma variável pública e faz um LED piscar.....	51
Figura 31: Sketch sem variável pública que faz um LED piscar.	52
Figura 32: Sketch que faz o LED piscar, usando o comando #define.....	53
Figura 33: Arduino IDE com botão do monitor serial destacado.....	54
Figura 34: Monitor serial, com a velocidade destacada.	55
Figura 35: Sketch que executa uma soma e mostra a conta no Monitor serial.	56
Figura 36: Monitor serial após rodar o sketch da Figura 35.	57
Figura 37: Sketch que usa a função condicional if, seguido de else.	58
Figura 38: Sketch que usa o comando de repetição while para fazer um LED piscar.	60
Figura 39: Sketch que usa o comando de repetição for para fazer LEDs piscarem.	61
Figura 40: Circuito com um push button.	63
Figura 41: Sketch usado para verificar se o botão está apertado.....	64
Figura 42: Circuito com botão em PULL UP.	66
Figura 43: Sketch contendo botão com INPUT_PULLUP.....	67
Figura 44: Potenciômetro, com indicação de onde conectar seus terminais.	69
Figura 45: Circuito com LDR ligado ao pino A0.....	70

Figura 46: Pulso.	71
Figura 47: sketch usando o comando map.	72
Figura 48: Circuito com capacitor junto a push button.	74
Figura 49: Primeira tentativa de debounce.	75
Figura 50: Segunda tentativa de debouncing.	77
Figura 51: Declaração da função chamada “Funcionalidade1”.	79
Figura 52: Sketch onde se utiliza uma função criada pelo próprio usuário.	80
Figura 53: Sketch para controlar servo motor com um potenciômetro	82
Figura 54: Programa que roda o motor nos ângulos 60, 120, 180 e 0°	83
Figura 55: Programa usando Display LCD	835

Índice de Tabelas

Tabela 1: Tabelas E e OU, respectivamente.	27
Tabela 2: Tabela de faixas dos resistores.....	40

Sumário

Capítulo 1 – O ARDUINO E SUAS APLICABILIDADES	8
Capítulo 2 – LÓGICA DE PROGRAMAÇÃO.....	18
2.1 ALGORITMOS E O PORTUGUÊS ESTRUTURADO	18
2.2 NOME DE VARIÁVEIS.....	21
2.3 OPERADORES ARITMÉTICOS E A ORDEM DE PRECEDÊNCIA	22
2.4 CONDICIONAIS	23
2.5 OPERADORES DE COMPARAÇÃO OU RELACIONAIS.....	25
2.6 OPERADORES LÓGICOS	26
2.7 ESTRUTURAS DE REPETIÇÃO	28
2.8 CONSIDERAÇÕES FINAIS SOBRE PORTUGUÊS ESTRUTURADO	31
2.9 EXERCÍCIOS	32
Capítulo 3 – NOÇÕES DE CIRCUITOS ELÉTRICOS APLICADAS AO ARDUINO	33
3.1 LEIS DE OHM.....	34
3.2 EXEMPLOS	35
3.3 CIRCUITOS EM SÉRIE.....	36
3.4 CIRCUITOS EM PARALELO.....	37
3.5 RESISTORES DO ARDUINO.....	38
3.6 ENTENDENDO A PROTOBOARD	40
3.7 ENTENDENDO O ARDUINO	43
Capítulo 4 – LINGUAGEM DE PROGRAMAÇÃO PARA ARDUINO.....	44
4.1 AMBIENTE DE DESENVOLVIMENTO INTEGRADO (IDE).....	44
4.2 ESTRUTURA BÁSICA DO <i>SKETCH</i>	46
4.3 ACENDENDO E APAGANDO UM LED	48
4.4 HIGH/LOW, 0/1, TRUE/FALSE	49
4.5 VARIÁVEIS NA PROGRAMAÇÃO DO ARDUINO	49
4.6 VARIÁVEIS PÚBLICAS E PRIVADAS.....	50
4.7 #DEFINE	53
4.8 O MONITOR SERIAL.....	54
4.9 OPERADORES, CONDICIONAIS	57
4.10 FUNÇÕES DE REPETIÇÃO	59
4.11 PINMODE	62
4.12 USANDO UM PUSH BUTTON.....	62
4.12.1 Pull up e Pull down.....	65

4.12.2 Comando INPUT_PULLUP	66
4.12.3 Comando DigitalRead().....	68
4.13 USANDO POTENCIÔMETRO E RESISTORES.....	68
4.14 USANDO UM LDR.....	69
4.14.1 Portas analógicas e digitais.....	70
4.14.2 Portas PWM.....	71
4.14.3 Comando map.....	72
4.15 CAPACITORES.....	73
4.15.1 Por que usar capacitores?	73
4.15.2 Push button e o efeito bouncing	73
4.16 FUNÇÕES	78
4.17 SERVO MOTOR E BIBLIOTECAS	81
4.18 DISPLAY LCD	84
4.19 CONSIDERAÇÕES FINAIS SOBRE PROGRAMAÇÃO PARA ARDUINO	87
CADERNO/RASCUNHO.....	88

Capítulo 1

O ARDUINO E SUAS APLICABILIDADES

A presente apostila objetiva ensinar a criar projetos básicos com Arduino, envolvendo conhecimentos sobre os circuitos elétricos e programação. Entretanto, antes de montar circuitos com diversos componentes eletrônicos e de utilizar uma linguagem de programação, é necessário entender o que é o Arduino.

De acordo com o site <http://www.arduino.cc>, "essa tecnologia é uma plataforma de código aberto de prototipagem eletrônica". Resumidamente, Arduino é uma placa, com um **microcontrolador**, ou um pequeno computador, que pode ser programado na linguagem C++ (que foi criada na década de 80), com pinos analógicos e digitais, conectável a um computador pessoal, que alimentará o circuito elétrico. Também faz parte do Arduino o *Software* onde serão feitos os *sketchs* (programas). Esse *Software* é chamado de IDE Arduino (*Integrated Development Environment* ou Ambiente de Desenvolvimento Integrado). Seu download pode ser feito no site www.arduino.cc. O Arduino existe há muito tempo, podendo ser usado para fazer diversos tipos de projetos, mostrados a seguir.

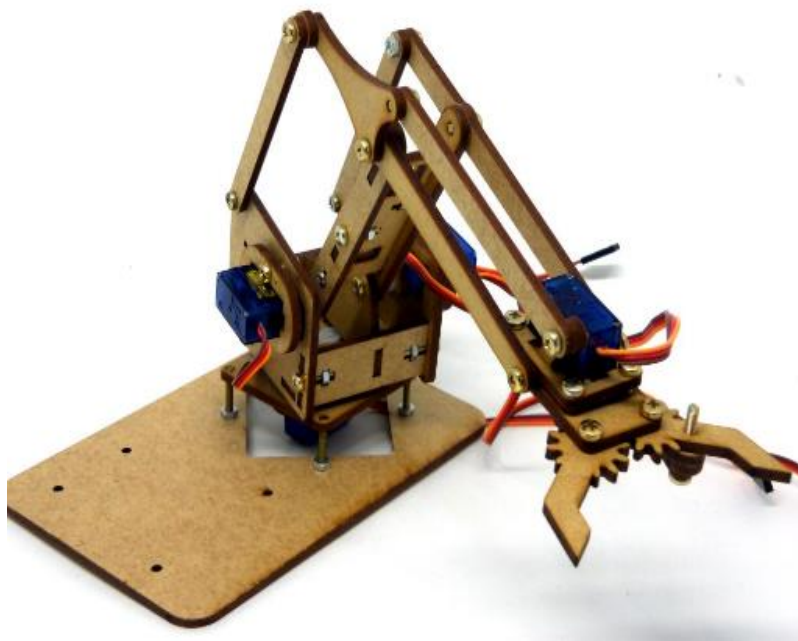


Figura 1: Braços robóticos

Fonte: <https://www.google.com/imghp?hl=pt-pt>



Figura 2: Sensores - pHmetro.

Fonte: <https://www.google.com/imghp?hl=pt-pt>



Figura 3: Sensores - sensor de gás.

Fonte: <https://www.google.com/imghp?hl=pt-pt>

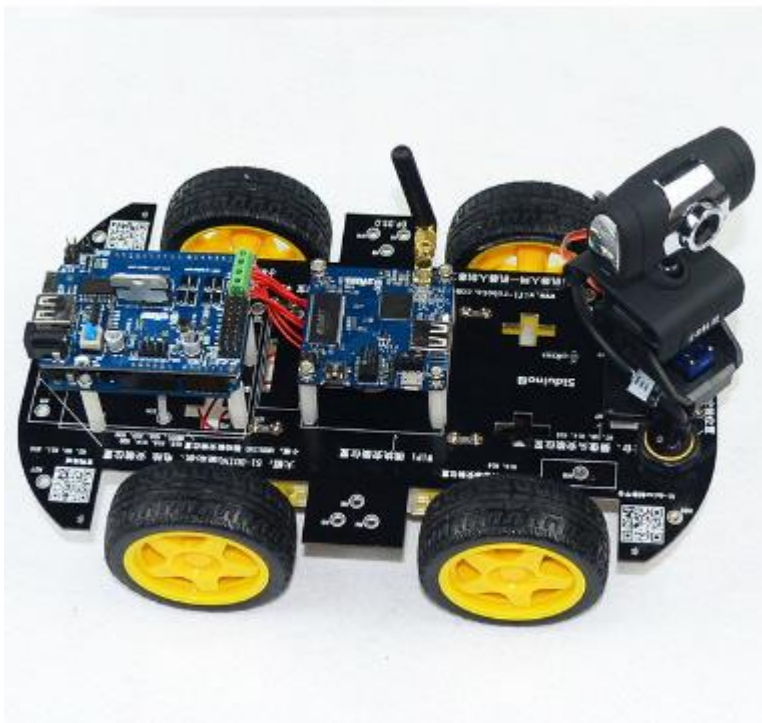


Figura 4: Carros de controle remoto.

Fonte: <https://www.google.com/imghp?hl=pt-pt>

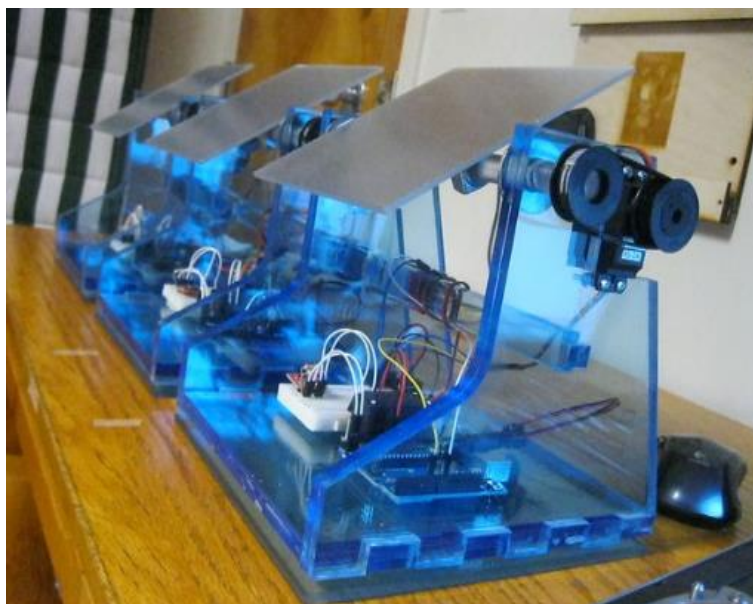


Figura 5: Painéis solares.

Fonte: <https://www.google.com/imghp?hl=pt-pt>



Figura 6: Automação Residencial.

Fonte: <http://twixar.me/cwR1>

A utilização da plataforma apresenta como uma de suas maiores vantagens o baixo custo da placa Arduino e seus componentes, possibilitando criar variados projetos de eletrônica que, de forma diferente, são muito custosos.

Somado ao baixo custo, existe uma grande variedade de componentes eletrônicos compatíveis com o Arduino, que conferem à plataforma grande versatilidade. Seguem alguns componentes:

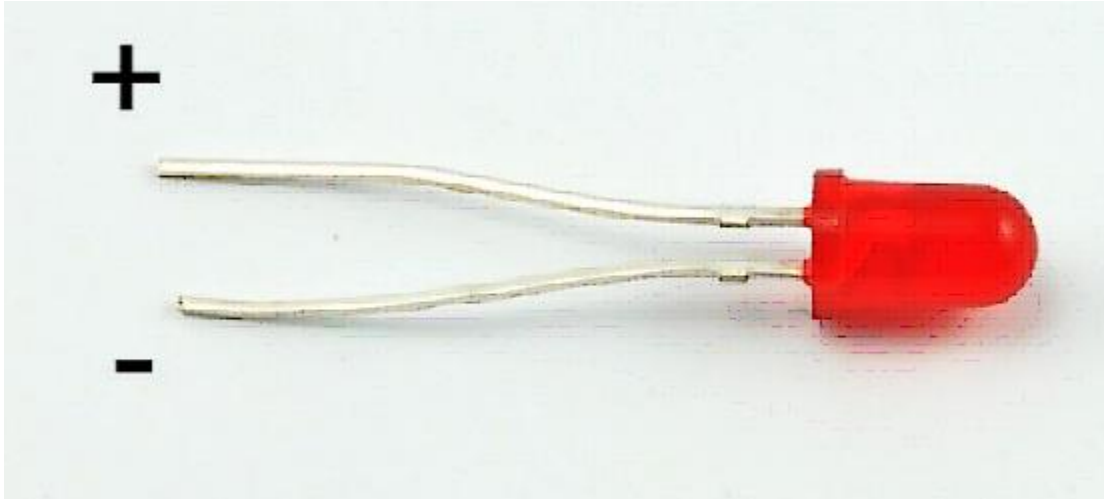


Figura 7: *Light Emitting Diode (LED).*

Fonte: <https://www.google.com/imghp?hl=pt-pt>



Figura 8: Push Button.

Fonte: <https://www.google.com/imghp?hl=pt-pt>



Figura 9: *Light Dependent Resistor* (LDR).

Fonte: <https://www.google.com/imghp?hl=pt-pt>



Figura 10: Potenciômetro.

Fonte: <https://www.google.com/imghp?hl=pt-pt>



Figura 11: Cabo Jumper.

Fonte: <https://www.google.com/imghp?hl=pt-pt>

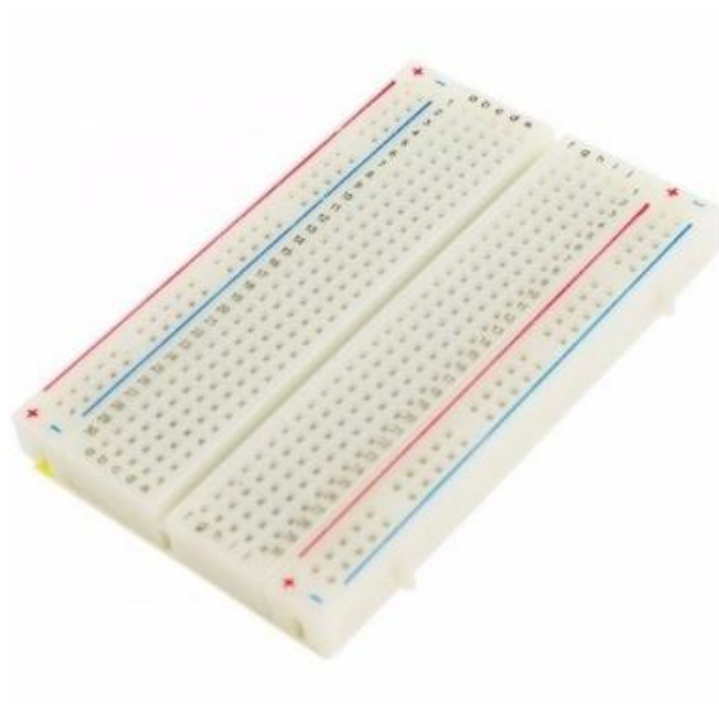


Figura 12: Protoboard.

Fonte: <https://www.google.com/imghp?hl=pt-pt>



Figura 13: Resistores.

Fonte: <https://www.google.com/imghp?hl=pt-pt>

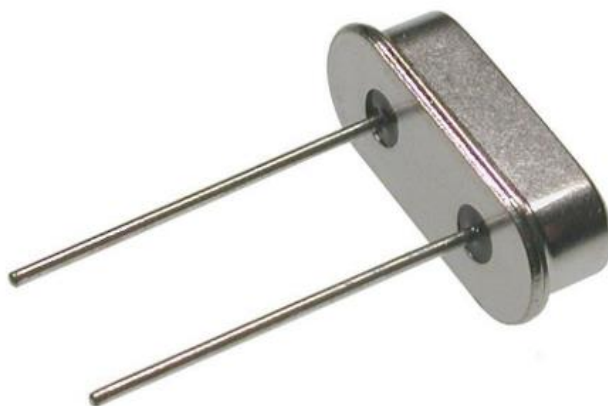


Figura 14: Cristal.

Fonte: <https://www.google.com/imghp?hl=pt-pt>



Figura 15: Capacitor.

Fonte: <https://www.google.com/imghp?hl=pt-pt>

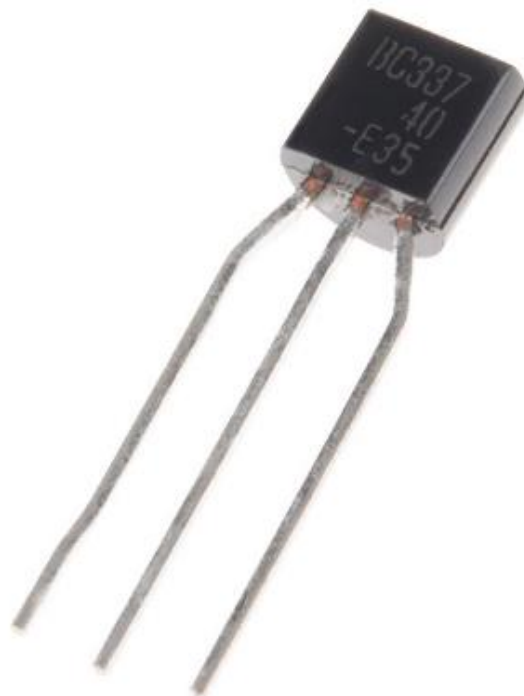


Figura 16: Transistor.

Fonte: <https://www.google.com/imghp?hl=pt-pt>



Figura 17: Servo Motor.

Fonte: <https://www.google.com/imghp?hl=pt-pt>



Figura 18: Sensor de aproximação.

Fonte: <https://www.google.com/imghp?hl=pt-pt>

Para construir circuitos elétricos usando componentes eletrônicos disponíveis é necessário ter noções básicas de elétrica (Capítulo 3) e para programar o microprocessador é necessário conhecer a linguagem de programação C++ (Capítulo 4). Antes de programar, no entanto, é preciso entender a lógica de programação (Capítulo 2).

Capítulo 2

LÓGICA DE PROGRAMAÇÃO

2.1 ALGORITMOS E O PORTUGUÊS ESTRUTURADO

Para entender a **lógica de programação**, primeiramente é necessário apresentar alguns conceitos. O termo refere-se à maneira de se escrever um algoritmo, que é um conjunto de passos finitos e organizados que, quando executados, resolvem determinados problemas. A lógica organiza os passos, de um algoritmo, a serem seguidos para atingir um objetivo. Se não for funcional, seguindo a definição dada, não é um algoritmo. Vale ressaltar ainda que a repetição de padrões é chamada de **rotina**. Assim, se usa a lógica de programação para criar um algoritmo que, fazendo uso de rotinas específicas, consegue resolver um problema determinado.

Inicialmente, não será usada nenhuma linguagem de programação neste curso. No lugar de uma linguagem, será usado o **português estruturado** (também chamado de Portugol), com intuito de resolver problemas hipotéticos. O termo refere-se a forma de utilizar a língua portuguesa, de maneira simplificada e próxima às linguagens de programação. O português estruturado não pode ser usado para programar o Arduino. Sua utilização visa acostumar os alunos com a lógica de programação, sendo este um estudo que precede o aprendizado de uma linguagem de programação propriamente dita. Nesta etapa da apostila, parte do conteúdo estará embasado em um curso *online* gratuito de lógica e algoritmos, disponível no link <https://www.cursoemvideo.com/course/curso-de-algoritmos/>.

Assim, a estrutura básica de um algoritmo, utilizando o português estruturado, é:

```
Algoritmo Nome           // É o nome do algoritmo
var                       // É a área onde se declara variáveis

inicio                   // É a área onde se coloca o corpo do programa

finalgoritmo             // Indica o fim do algoritmo
```

O português estruturado, apesar de não ser uma linguagem de programação, possui suas regras e sua própria sintaxe. Nesta apostila, estas regras serão adaptadas para que fiquem o mais

próximo da linguagem de programação e dos comandos usados no Arduino. O objetivo do curso não é se aprofundar em português estruturado ou lógica de programação, e sim aprender a programar e criar projetos básicos com Arduino.

Para criar um programa, usando português estruturado, que mostre na tela do computador a frase "Olá Mundo"? O programa (ou melhor, o *sketch*) deve ficar assim:

```
Algoritmo sketch1
var
inicio
    Escreva ("Olá Mundo");
finalgoritmo
```

No exemplo acima não há necessidade de declarar nenhuma variável, somente de mostrar uma mensagem no monitor. No português estruturado adaptado que está sendo usado, sempre que se for mostrar uma mensagem, a mesma deve se encontrar entre aspas e dentro de parênteses. Além disso, o final de todo comando deve ser indicado por ";" .

É importante aprender a criar um *sketch* que utilize alguma variável. Variáveis são como espaços (ou "gavetas") na memória do computador, onde se armazena algum valor. Os valores podem ser números, letras, palavras, frases, ou até mesmo valores lógicos, como verdadeiro ou falso. Para armazenar um valor dentro de uma variável, é necessário criar a variável (declará-la) e indicar que tipo de valor será armazenado dentro dela. Por exemplo, para se armazenar um nome deve-se declarar uma variável do tipo "caractere".

Como se pode declarar uma variável, atribuir a ela um nome próprio e então mostrar no monitor o nome guardado nesta variável? Veja a resolução a seguir:

Algoritmo Sketch2

```
var
    caractere nome;
inicio
    nome = "Hugo",
    Escreva (nome);
finalgoritmo
```

Ao executar o *sketch* acima, deve aparecer no monitor o valor da variável "nome", que é "Hugo".

Como se pode perguntar o nome do usuário, armazenar esse valor dentro de uma variável e mostrar a mensagem "Bom dia, <nome>" no monitor? Para isso é necessário usar o comando "Leia" para receber informações do usuário. Veja:

Algoritmo Sketch3

```
var
    caractere nome;
inicio
    Escreva ("Digite seu nome: ");
    Leia (nome);
    Escreva ("Bom dia ");
    Escreva (nome);
finalgoritmo
```

Da mesma forma que a variável "nome" do exemplo acima foi declarada como "caractere", para criar variáveis que armazenam números inteiros deve-se usar o tipo "inteiro".

Assim, o quarto *sketch* visa ler dois valores inteiros, armazená-los em duas variáveis e mostrar no monitor a soma entre eles:

Algoritmo Sketch4

```
var
    inteiro numero1;
    inteiro numero2;
    inteiro soma;
inicio
    Escreva ("Digite o primeiro numero: ");
    Leia (numero1);
    Escreva ("Digite o segundo numero: ");
    Leia (numero2);
    soma = numero1 + numero2;
    Escreva ("A soma entre eles é: ");
    Escreva (soma);
finalgoritmo
```

2.2 NOME DE VARIÁVEIS

Um detalhe importante para o qual deve-se sempre atentar é o nome das variáveis que são declaradas. Existem algumas regras. Primeiramente, as variáveis não podem ter seu nome começando por um número, mas sim ter um número no seu nome. Além disso, não se pode usar caracteres especiais no nome das variáveis (tais como "@, &, ?, !, #"). O caractere *underline* (_) pode ser usado, mas não a barra de espaço. Por fim, palavras reservadas não podem ser usadas como nome de variáveis. As palavras reservadas são comandos usados no português estruturado, tais como "Leia" e "Escreva". Assim, os nomes abaixo não **podem** ser usados como nomes de variáveis:

1numero	alun@	alunos e alunas
inicio	Escreva	ele_&_ela

Exemplos de palavras que podem ser usadas como nome de variáveis seriam:

numero1	aluno	alunos_e_alunas
InicioNumero	EscrevaValor	ele_e_ela

2.3 OPERADORES ARITMÉTICOS E A ORDEM DE PRECEDÊNCIA

Voltando ao *sketch4*, nota-se que foi feita uma soma entre dois números inteiros. Para realizar essa operação matemática foi utilizado, naturalmente, o sinal "+". Para realizar uma subtração, usa-se o sinal "-". Para realizar uma multiplicação, no entanto, não se usa o "x". Nesse caso, é empregado o "*". A divisão é representada pelo operador "/". Para se obter o módulo de um valor, se utiliza o sinal "%", que representa, nesse caso, o resto de uma divisão. Dessa forma, o resultado da conta $5 \% 2$ seria 1. Um último operador aritmético é a exponenciação, representada pelo sinal "^". Assim, os símbolos apresentados são os principais **operadores aritméticos** usados, não esquecendo do sinal "=", que é o operador de atribuição.

Se necessário fazer uma conta mais elaborada, envolvendo mais de um operador, é importante saber a ordem em que as operações aritméticas serão executadas pelo computador. As primeiras contas que são feitas encontram-se sempre dentro de parênteses. Em seguida, se dá preferência para as exponenciações (^). Depois das exponenciações, as divisões (/) e multiplicações (*) são feitas na ordem em que aparecem. Por fim, se executa as adições (+) e subtrações (-). Para conhecer mais sobre operadores aritméticos e como usá-los, em problemas usando português estruturado, é recomendável assistir à vídeo aula disponível no link <https://www.cursoemvideo.com/unit/curso-de-algoritmos-03-comando-de-entrada-e-operadores/?id=973> (terceira aula do curso de algoritmos e lógica).

A estrutura básica do português estruturado está apresentada. Relembrando: caso se crie uma variável que vá receber uma letra, ou uma sequência de letras formando um nome, esta deverá ser do tipo "caractere", e o nome que será armazenado deverá estar entre aspas. Caso se crie uma variável que receba um número inteiro, esta deverá ser do tipo "inteiro". Agora, caso se crie uma variável que receba um número decimal, como 1,5 ou 2,3, esta deverá ser do tipo "real". Sabendo disso, vamos à próxima etapa deste capítulo, que trata de condicionais no português estruturado.

2.4 CONDICIONAIS

Para executar determinada ação caso uma condição seja satisfeita, usamos o comando "se". Por exemplo, se o objetivo é ler um número inteiro e mostrá-lo no monitor, e se o número for maior que 10, pode-se usar o *sketch*:

Algoritmo Sketch5

```
var
    inteiro n;
inicio
    Escreva ("Digite o numero: ");
    Leia (n);
    se (n > 10){
        Escreva ("O valor digitado é ");
        Escreva (n);
    }
finalgoritmo
```

Note a estrutura da condicional: o comando "se", seguido da condição entre parênteses (que a variável n seja maior que 10), seguido de chaves. Todos os comandos dentro das chaves serão executados somente se a condição dentro dos parênteses for satisfeita.

Caso o número digitado não seja maior que dez, deve-se mostrar uma mensagem no monitor dizendo que o número é pequeno demais, e caso seja maior, deve-se mostrar o valor digitado no monitor. O *sketch6* mostra como resolver o problema:

Algoritmo Sketch6

```
var
    inteiro n;
inicio
    Escreva ("Digite o numero: ");
    Leia (n);
    se (n > 10){
        Escreva ("O valor digitado é ");
```

```

        Escreva (n);
    }senao{
        Escreva ("O valor digitado é pequeno demais");
    }
fimalgoritmo

```

Se a condição dentro dos parênteses, após o comando "se", for atendida, tudo o que está dentro das chaves do "se" será executado; se não, tudo o que está dentro das chaves do comando "senao" será executado. Alguns problemas não apresentam somente duas opções, e é justamente quando isso acontece que se explora ao máximo os comandos "se" e "senao", colocando uma condicional dentro de outra condicional, criando assim inúmeros caminhos que o programa poderá seguir. Observe o *sketch7*, que contém algumas condicionais, além de operadores que ainda não foram apresentados:

Algoritmo Sketch7

```

var
    inteiro n1;
    inteiro n2;
    inteiro soma;
inicio
    Escreva ("Digite o primeiro numero, inteiro e positivo: ");
    Leia (n1);
    Escreva ("Digite o segundo numero, inteiro e positivo: ");
    Leia (n2);
    soma = n1 + n2;
    se (soma >= 0 && soma < 10){
        Escreva ("A soma é menor que dez ");
    }senao{
        se (soma >= 10 && soma < 20){
            Escreva ("O valor é maior ou igual a dez e menor que vinte");
        }senao{
            Escreva ("O valor digitado é maior ou igual a vinte");
        }
    }
fimalgoritmo

```


O *sketch7* pede para o usuário digitar dois números, ambos inteiros e positivos. Depois disso, são somados os dois valores e armazenado o resultado em uma variável inteira chamada "soma". Caso a soma dos dois números seja menor que dez, o programa mostrará a mensagem "A soma é menor que dez" no monitor, e se a soma for um valor compreendido entre dez e vinte, o programa mostrará no monitor a mensagem "O valor é maior ou igual a dez e menor que vinte". Se nenhuma das condições anteriores for satisfeita, o programa mostrará no monitor a mensagem "O valor digitado é maior ou igual a vinte".

Todos os comandos que estão dentro de um "se" ou de um "senao" estão um pouco afastados para a direita, e dentro das chaves da respectiva condicional. Perceba que foram usados sinais (\geq e $<$) para comparar o valor da variável "soma" com outros valores numéricos. Os sinais são os operadores de comparação, e serão explicados nos tópicos a seguir, juntamente com a utilização do sinal " && ", que é um operador lógico.

2.5 OPERADORES DE COMPARAÇÃO OU RELACIONAIS

Seguem abaixo alguns sinais, chamados de operadores de comparação. São muito usados dentro de comandos condicionais, para comparar valores numéricos.

>	maior	<=	menor ou igual
<	menor	==	igual
>=	maior ou igual	!=	diferente

Uma atenção especial deve ser dada ao operador " $==$ ". Um erro muito comum é considerar o sinal " $=$ " como "igual", sendo que o sinal correto seria " $==$ ". O sinal " $=$ " deve ser interpretado como "recebe". Assim, sempre que se quer atribuir determinado valor a uma variável, deve-se dizer que a variável recebe o valor, utilizando-se o sinal " $=$ ". Para igualar duas variáveis, usa-se o sinal " $==$ ". Olhe o *sketch8* como exemplo:

Algoritmo sketch8

var

inteiro x;

inicio

x = 12; //variável x recebe valor 12

se (x == 12){ //se variável x for igual a 12

Escreva (" o numero é igual a 12");

}

fimalgoritmo

2.6 OPERADORES LÓGICOS

Os últimos operadores que serão abordados são os operadores lógicos. Seguem abaixo exemplos:

! não

&& e

|| ou

Os operadores podem ser usados em diversas situações. No próprio *sketch7* pode ser encontrado o comando se (soma >= 10 && soma < 20){.

Se a variável "soma" for maior ou igual a 10 e for menor que 20, os comandos dentro da condicional serão executados. O && foi usado para colocar duas condições para que ocorram os comandos. Assim, o programa só acionará os comandos se ambas as condições forem aceitas.

De maneira semelhante ao && temos o | |, que significa " ou ", e é encontrado com frequência dentro de condicionais. Observe o *sketch 9*:

Algoritmo Sketch9

var

```

inteiro n1;
inteiro n2;
inteiro soma;
inicio
    Escreva ("Digite o primeiro numero, inteiro e positivo: ");
    Leia (n1);
    Escreva ("Digite o segundo numero, inteiro e positivo: ");
    Leia (n2);
    soma = n1 + n2;
    se (soma >= 50 || soma <= 10){
        Escreva ("A soma deu um valor muito alto ou muito baixo ");
    }
finalgoritmo

```

No exemplo acima, se a soma dos dois números for maior que 50 **ou** menor que 10, aparecerá no monitor a mensagem "A soma deu um valor muito alto ou muito baixo". Observe que ambas as condições não precisam ser satisfeitas para que o comando seja executado, basta uma.

Tanto o operador && como o operador || possuem uma tabela, que exemplifica como funcionam, qual a lógica envolvida em usar && e ||. Observe:

Tabela 1: Tabelas E e OU, respectivamente.

p	q	p E q	p	q	p OU q
V	V	V	V	V	V
V	F	F	V	F	V
F	V	F	F	V	V
F	F	F	F	F	F

Fonte: <https://www.cursoemvideo.com/>.

A primeira tabela é referente ao operador &&, e podemos chamá-la de tabela E. Nessa tabela, "p" e "q" representam duas premissas, e se uma e a outra forem verdadeiras, então p && q

é verdadeiro também, como pode ser visto na primeira linha da tabela. Se uma das premissas (p ou q) for falsa, $p \text{ \&\& } q$ também será falso.

A segunda tabela é referente ao operador $||$, e podemos chamá-la de tabela OU. Nessa tabela, se uma das premissas ($p \text{ ou } q$) for verdadeira, ou ambas o forem, $p || q$ será verdadeiro. Só teremos $p || q$ falso se ambas as premissas forem falsas. As duas tabelas, do E e do OU, são chamadas de Tabelas Verdades.

Caso deseje saber mais sobre operadores lógicos e relacionais, é recomendável que assista o vídeo disponível no link <https://www.cursoemvideo.com/unit/curso-de-algoritmos-04-operadores-logicos-e-relacionais/?id=973>.

2.7 ESTRUTURAS DE REPETIÇÃO

Sempre que se quer fazer um *sketch* que execute um comando, ou uma sequência de comandos, várias vezes ou um número certo de vezes, deve-se usar estruturas de repetição. Esse tipo de comando permite que uma parte do programa fique sendo executada repetidamente, até que alguma condição seja satisfeita.

Para isso, pode-se usar o comando "enquanto". Esse comando funciona da seguinte forma: enquanto uma condição não for satisfeita, executa-se algum comando. Veja o *sketch10*:

Algoritmo Sketch10

var

inteiro n1;

inteiro c = 1;

inteiro s = 0;

inicio

enquanto (c <= 5){

Escreva ("Digite um numero, inteiro e positivo: ");

Leia (n1);

s = s + n1;

c = c + 1;

}

fimalgoritmo

No exemplo acima, enquanto a variável c for menor ou igual a 5, o programa lê um número ($n1$) e adiciona-o na variável s . A variável c é então acrescida de 1 e, caso seja menor ou igual a 5, o ciclo se repete. Quando a variável c se tornar maior que 5, o ciclo é interrompido.

Outro tipo de estrutura de repetição que pode ser usada é o "para". Nesse caso, se o objetivo for escrever um *sketch* que faça a mesma coisa que o *sketch10*, procede-se da seguinte forma:

Algoritmo Sketch11

var

inteiro $n1$;

inteiro c ;

inteiro $s = 0$;

inicio

para($c = 1$; $c \leq 5$; $c = c + 1$){

Escreva ("Digite um numero, inteiro e positivo: ");

Leia ($n1$);

$s = s + n1$;

}

fimalgoritmo

No caso acima, o valor inicial da variável c é 1 ($c = 1$), os comandos serão executados enquanto c for menor ou igual a 5 ($c \leq 5$), e o valor de c será acrescido de 1 toda vez que os comandos forem executados ($c = c + 1$). Todas essas informações se encontram dentro dos parênteses que seguem o comando "para".

Tanto o "para" como o "enquanto" servem nesse caso, e existem muitas semelhanças entre os dois comandos. Observe o comando "enquanto" seguido do comando "para":

```

inteiro n1;                                //roxo: comandos
inteiro c = 1;                             //vermelho: condição de parada
inteiro s;                                 //verde: progressão

//azul: valor inicial de c

```

```

enquanto (c <= 5){
    Escreva ("Digite um numero, inteiro e positivo: ");
    Leia (n1);
    s = s + n1;
    c = c + 1;
}

```

```

inteiro n1;
inteiro c;
inteiro s;
para(c = 1; c <= 5; c = c + 1){
    Escreva ("Digite um numero, inteiro e positivo: ");
    Leia (n1);
    s = s + n1;
}

```

Ambos os comandos acima leem um número cinco vezes e salvam a soma entre os números lidos dentro da variável *s*. A lógica dos comandos é diferente, mas o resultado é o mesmo. Acima, pode-se ver que algumas partes do código são realocadas quando usado "enquanto" e se muda para o comando "para", e vice-versa. As partes iguais estão com as mesmas cores. Portanto, para mudar de um comando para o outro, bastaria realocar as partes do código em função do comando que está sendo usado.

Assim, uma fórmula geral para o comando "para" seria:

```
para(valor inicial de c; condição de parada; progressão){  
    comandos;  
}
```

No caso do comando "enquanto", seria:

```
valor inicial de c;  
enquanto(condição de parada){  
    comandos;  
    progressão;  
}
```

2.8 CONSIDERAÇÕES FINAIS SOBRE PORTUGUÊS ESTRUTURADO

O português estruturado apresentado neste capítulo foi modificado para se aproximar ao máximo da forma como se programa o Arduino. Assim, é importante frisar que muitos dos comandos apresentados aqui não são usualmente empregados no português estruturado. Por exemplo, quando se deseja atribuir um valor a uma variável, no português estruturado usa-se o sinal "<=", mas nesta apostila foi usado o sinal "=". Isso foi feito propositalmente, uma vez que na linguagem C++ o sinal utilizado para atribuir um valor a uma variável é de fato o " = ".

Não obstante, saber as diferenças é importante, caso o aluno se depare com algoritmos que estejam escritos de forma diferente da ensinada aqui.

Outro ponto que deve ser ressaltado é a importância do site <https://www.cursoemvideo.com/> para a formulação deste capítulo. O referido site disponibiliza diversos cursos gratuitos, incluindo um curso de Lógica de Programação, que foi usado para escrever parte desta apostila. O citado curso pode e deve ser usado para reforçar mais ainda seus conhecimentos nesta disciplina. Ao longo do capítulo foram citadas algumas videoaulas de Lógica que podem ser assistidas para um maior conhecimento sobre o assunto. Essas aulas contribuem muito para o entendimento da matéria e, além disso, caso o aluno se cadastre no site e assista à todas as aulas, poderá fazer uma prova ao

final e, se a média for superior a 7, terá direito à um certificado reconhecido pelo Ministério da Educação e Cultura. Relembrando: o português estruturado é ligeiramente diferente daquele apresentado aqui.

Para treinar mais ainda o português estruturado, é válido baixar o programa Visualg, disponível no link <http://visualg3.com.br/>, que permite ao usuário colocar seus códigos, em português estruturado, e ver o programa rodar. É um ótimo treinamento para ver como seus *sketchs* funcionam e se estão funcionando corretamente.

2.9 EXERCÍCIOS

- 1) Escreva um algoritmo que pergunte ao usuário seu nome e depois mostre a seguinte mensagem no monitor: Olá, <nome>!
- 2) Escreva um algoritmo que leia três números e mostre a soma dos dois primeiros, a soma dos dois últimos, e a soma de todos os três números.
- 3) Escreva um algoritmo que peça ao usuário para escrever seu nome e, caso o nome seja "João", "Maria" ou "Pedro", mostre a seguinte mensagem no monitor: "Seu nome é bonito!". Caso não seja nenhum dos nomes, mostre a mensagem: Prazer, <nome>!
- 4) Escreva um algoritmo que peça ao usuário para digitar dois números e mostre no monitor a média entre eles e diga qual o maior e qual o menor.
- 5) Escreva um algoritmo que leia três números e mostre-os no monitor em ordem decrescente.
- 6) Escreva um algoritmo que leia dois números e, se o primeiro for maior que o segundo, mostre no monitor a média entre eles; caso contrário, mostre a soma entre eles.
- 7) Escreva um algoritmo que leia um número e diga se é par ou ímpar.
- 8) Escreva um algoritmo que peça ao usuário para digitar um número inteiro e, caso esse número não seja 5, continue demandando que seja digitado um número.

Capítulo 3

NOÇÕES DE CIRCUITOS ELÉTRICOS APLICADAS AO ARDUINO

Para criar os projetos variados com Arduino é imprescindível possuir uma base de física, especificamente de circuitos elétricos. A programação que será feita, em C++, deve estar compatível com o circuito elétrico montado. Caso contrário, mesmo que a sintaxe e a lógica do *sketch* estejam corretas, o projeto pode não funcionar devidamente por causa de um circuito mal montado.

Primeiramente, deve-se ter em mente que o circuito irá funcionar somente quando alimentado por uma fonte de energia, que será majoritariamente o computador e estará ligado à placa Arduino por um cabo USB.

Com a placa alimentada, se tem uma tensão, ou diferença de potencial. Este termo indica que há, em um ponto, uma carga positiva e, em outro ponto, uma carga negativa (igual a uma pilha, que tem um lado positivo e um negativo). O lado positivo se chama cátodo e o negativo se chama ânodo. Para que qualquer circuito funcione é necessário que esteja fechado, ou seja, que o lado carregado positivamente esteja ligado ao lado carregado negativamente. Quando isso ocorre, a tensão flui através do circuito, alimentando todos os componentes eletrônicos que estiverem no caminho.

Convencionalmente se diz que a corrente flui do polo positivo para o polo negativo. No entanto, o polo negativo é o que possui grande quantidade de elétrons, de forma que o sentido real da corrente é o caminho percorrido pelos elétrons, do negativo em direção ao positivo.

De qualquer forma, essa variável (tensão), apresentada acima, é quantificada em Volts, e é representada pela letra V. Alguns eletrodomésticos utilizam 110 Volts, enquanto outros utilizam 220 Volts. Quando alimentado um aparelho de 110 Volts com 220 Volts o aparelho queima. O mesmo ocorre com os componentes eletrônicos do Arduino, ou com o próprio Arduino.

A placa utiliza 5 V para funcionar, e cada componente eletrônico utiliza uma tensão específica. Os LEDs, por exemplo, utilizam algo em torno dos 2 V. Se alimentado um componente com uma tensão muito alta, pode-se queimar. Para evitar inconvenientes, quando se sabe que a

tensão está alta demais para determinado componente, coloca-se um resistor no circuito, o que abaixa a tensão. A unidade de medida da resistência é o Ohm (Ω), representada pela letra R.

Uma última variável que precisa ser apresentada é a Intensidade da corrente, que representa a quantidade de carga elétrica que atravessa certo condutor elétrico, em um período de tempo determinado. Assim, para se calcular a intensidade da corrente, aplica-se a fórmula $I = Q/\Delta t$, onde i é a intensidade, Q é a carga elétrica e Δt é o intervalo de tempo. A unidade de medida de i é o Ampère, representado pela letra A.

3.1 LEIS DE OHM

Georg Simon Ohm, um físico alemão, muito contribuiu para a elétrica, tanto que a unidade de medida de resistência se chama Ohm, em sua homenagem.

A primeira lei de Ohm diz que a corrente elétrica, que passa em um componente elétrico, é diretamente proporcional à tensão. Assim, se a tensão aumenta, a intensidade da corrente também aumenta; se a tensão diminui, a intensidade também diminui.

A razão V/i é uma constante, a qual Ohm nomeou de Resistência (R). Desenvolvendo a fórmula pode-se chegar a sua forma mais conhecida:

$$V = R \cdot i$$

A segunda lei de Ohm apresenta a Resistividade (ρ), que é diretamente proporcional ao comprimento do objeto e inversamente proporcional à sua área. A resistividade é uma constante de cada material, variando somente com a temperatura, afetando a resistência (R).

Pode-se usar a fórmula $V = R \cdot i$ para calcular quantos ohms são necessários para evitar queimar os componentes do circuito. Uma vez calculada a resistência necessária, basta pegar o resistor correspondente e adicioná-lo ao circuito.

Além da tensão, da intensidade e da resistência, a potência também vale ser citada para que se tenha uma noção básica da elétrica envolvida nos circuitos do Arduino. O termo potência refere-se à quantidade de energia elétrica que se transforma em outro tipo de energia - como a energia luminosa, no caso dos LEDs - por uma unidade de tempo. A potência pode ser calculada

multiplicando a tensão pela intensidade da corrente. Assim, temos que $Pot = V.i$. Desenvolvendo a fórmula, pode-se substituir V por $R.i$, gerando a expressão $Pot = R.i^2$.

3.2 EXEMPLOS

Se uma corrente de 20mA passa por um resistor de $220\ \Omega$, qual a diferença de potencial?

R: A diferença de potencial é a tensão, representada por V , a qual é igual a resistência (220Ω) vezes a intensidade (20mA ou 0,02A). Assim:

$$V = 220 \cdot 20 \cdot 10^{-3}$$

$$V = 4,4\ V$$

Outro exemplo. Se tem uma tensão de 5 V entre os terminais de uma placa Arduino, juntamente com uma corrente de 50 mA. Qual a resistência entre os terminais?

R: A resistência pode ser calculada usando a fórmula $V = R.i$. Logo:

$$5 = R \cdot 50 \cdot 10^{-3}$$

$$(5/50).10^3 = R$$

$$R = 100\ \Omega$$

Último exemplo.

a) Se $V = 5V$ e $i = 50mA$, qual a resistência desse circuito?

b) Se, no mesmo circuito, troca-se a intensidade (i) para 500mA, qual a nova tensão (V)?

$$5 = R \cdot 50 \cdot 10^{-3}$$

$$R = 100\ \Omega$$

Há $100\ \Omega$ no circuito e, ao mudar a intensidade da corrente, de $50\ \text{mA}$ para $500\ \text{mA}$, a tensão será:

$$V = 100 \cdot 500 \cdot 10^{-3}$$

$$V = 50\ \text{V}$$

Observa-se aqui que a intensidade (i) é diretamente proporcional à tensão (V).

3.3 CIRCUITOS EM SÉRIE

Circuitos em série são aqueles em que os componentes se encontram em sequência, um atrás do outro. A resistência de cada componente se soma aos demais, criando-se assim uma resistência única para o circuito.

Este tipo de arranjo possui vantagens e desvantagens. É vantajoso pois permite colocar vários resistores em série, até que se atinja a resistência desejada. Por outro lado, é desvantajoso porque se um dos componentes do circuito queimar, o circuito fica aberto e, portanto, deixa de funcionar.

Observe abaixo um esquema de circuito em série:

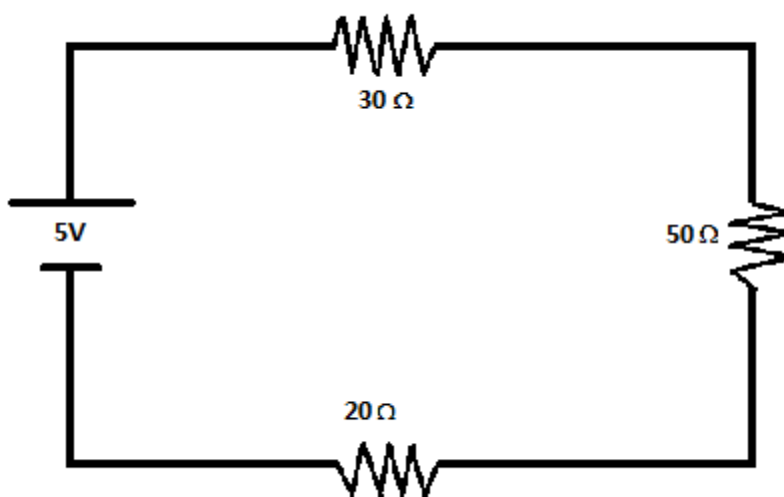


Figura 19: Circuito elétrico em série com três resistores.

Fonte: O Autor (2019)

A resistência total do circuito é $30 + 50 + 20 = 100 \, \Omega$. Assim, a corrente que atravessa o circuito é da ordem de 50 mA, visto que:

$$V = R \cdot i$$

$$5 = 100 \cdot i$$

$$i = 0,05 \, \text{A}, \text{ ou } 50 \, \text{mA}$$

3.4 CIRCUITOS EM PARALELO

Diferentemente dos circuitos em série, aqueles em paralelo não somam as resistências, mas as dividem proporcionalmente. A tensão elétrica (V) será a mesma em todas as cargas do circuito, mas as resistências (Ω) podem ser diferentes. Este tipo de arranjo é vantajoso, uma vez que mesmo que um dos componentes do circuito queime, o restante continuará funcionando. Por outro lado, como a tensão é a mesma em todos os ramos do circuito, a potência dissipada¹ será muito maior, configurando assim um gasto mais elevado do que um circuito em série.

Observe o circuito em paralelo abaixo:

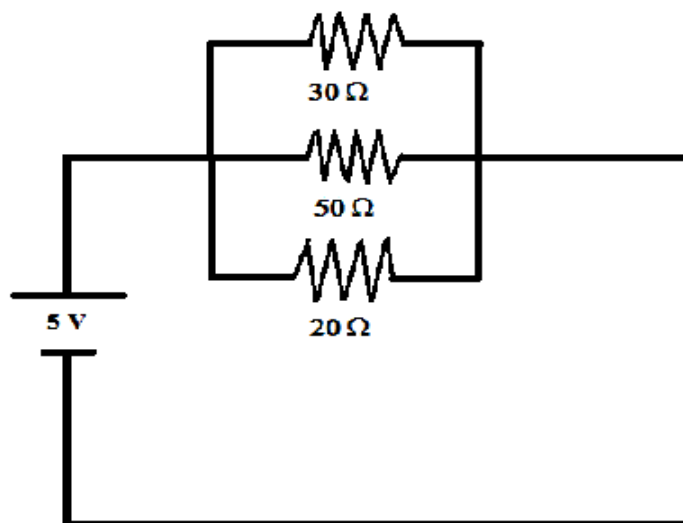


Figura 20: Circuito elétrico em paralelo com três resistores.

Fonte: O Autor (2019)

¹ Sendo resistores componentes que convertem energia elétrica em térmica, o termo potência dissipada refere-se à quantidade de calor que é transferida, por resistores, para o seu entorno, por uma unidade de tempo.

Para se calcular a resistência total do circuito, não basta somar as resistências. A fórmula usada para se calcular a resistência total é:

$$1/R = (1/R1) + (1/R2) + (1/R3) \dots + (1/Rn)$$

Assim, no exemplo dado, a resistência total seria:

$$1/R = (1/30) + (1/50) + (1/20)$$

$$1/R = 10/300 + 6/300 + 15/300$$

$$1/R = 31/300$$

$$R = 300/31 = 9,67 \, \Omega$$

Perceba que, utilizando os mesmos valores do primeiro exemplo (30, 20 e 50 Ω), obtém-se uma resistência total muito menor (aproximadamente 10 Ω , em comparação com os 100 Ω obtidos antes).

Sabendo que a resistência do circuito é 9,67 Ω , e que é alimentado por uma fonte de 5 Volts, é possível calcular a intensidade da corrente, usando a fórmula $V = R.i$, o que resulta em 0,517 A ou 517 mA.

3.5 RESISTORES DO ARDUINO

O Arduino funciona com uma tensão de 5 V e seus inúmeros componentes utilizam tensões distintas, muitas vezes menores do que esse valor. Para evitar queimar os componentes é necessário abaixar a tensão, adequando-a. Uma tensão muito baixa, no entanto, pode ser insuficiente para fazer funcionar determinado componente eletrônico.

Para o bom funcionamento do circuito, os resistores certos devem ser colocados de forma que a tensão se adapte aos elementos do circuito. Se é conhecida a tensão e a intensidade de corrente ideal para certo componente, basta aplicar a fórmula $V = R.i$ para saber qual resistor deve-se usar.

Os resistores vêm marcados com quatro pequenas faixas, coloridas, que representam o valor de sua resistência, conforme se observa na gravura a seguir:

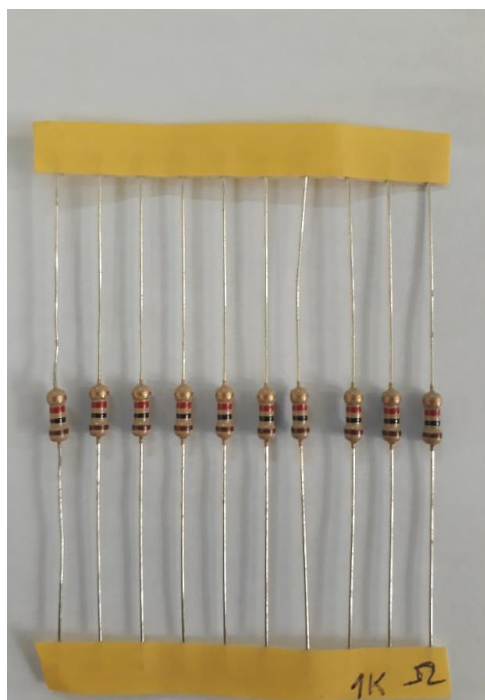


Figura 21: Resistores.

Fonte: O Autor (2019)

A primeira faixa representa a casa das dezenas, a segunda faixa representa a casa das unidades, a terceira faixa representa um multiplicador (10, 100, 1000, entre outros), a última faixa representa a faixa de tolerância, em %, do resistor.

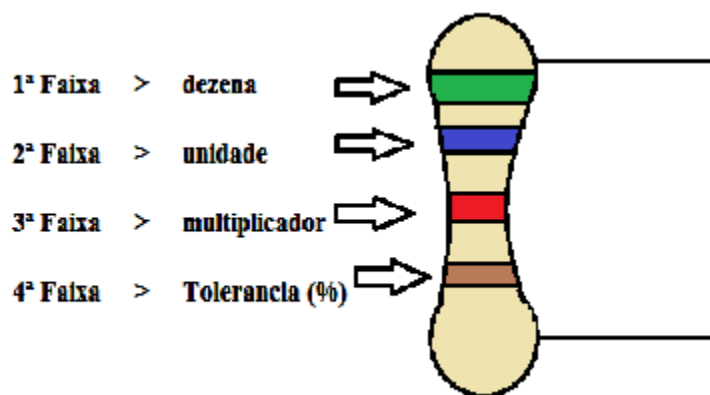


Figura 22: Código e padrão de cores dos resistores.

Fonte: O Autor (2019)

A tabela a seguir mostra como interpretar o padrão de cores dos resistores:

Tabela 2: Tabela de faixas dos resistores.

Cor	1ª Faixa	2ª Faixa	3ª Faixa	4ª Faixa
Preto	-	0	*1	-
Marrom	1	1	*10	1%
Vermelho	2	2	*100	2%
Laranja	3	3	*1000	3%
Amarelo	4	4	*10000	4%
Verde	5	5	*100000	-
Azul	6	6	*1000000	-
Violeta	7	7	-	-
Cinza	8	8	-	-
Branco	9	9	-	-
Prata	-	-	*0,01	10%
Dourado	-	-	*0,1	5%

Fonte: O Autor (2019)

Assim, em um resistor com faixas verde, azul, vermelha e marrom, respectivamente, se tem $5600\ \Omega$, ou $5,6\ \text{k}\Omega$, com margem de erro de 1%. Já resistores com faixas marrom, preto, vermelho e dourado, respectivamente, são resistores de $1000\ \Omega$, ou $1\ \text{k}\Omega$.

3.6 ENTENDENDO A PROTOBOARD

A protoboard é uma peça essencial para a prototipagem de qualquer projeto com Arduino, consistindo em uma placa com inúmeros pequenos furos, alguns ligados entre si formando fileiras. Ao energizar um dos furos, a fileira inteira fica energizada. Observe a figura a seguir:

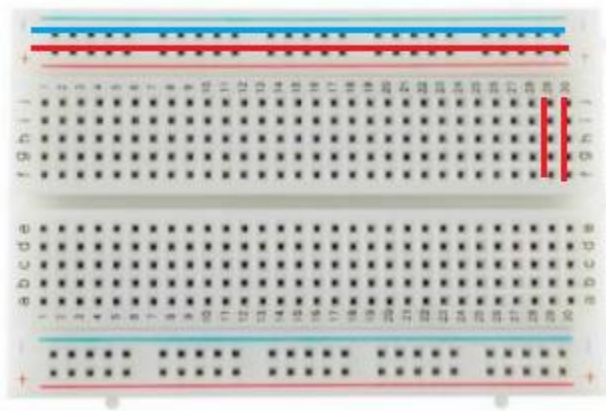


Figura 23: Protoboard.

Fonte: O Autor (2019)

As linhas vermelhas realçam os pinos que constituem fileiras. Na lateral da protoboard tem uma fileira maior, também pintada de vermelho, que muitas vezes é usada para se colocar uma tensão de 5 V. Também na lateral, se tem uma fileira longa que está pintada de azul, sendo esta comumente usada para ligar a protoboard ao pino GND (terra) do Arduino.

As fileiras são paralelas e, portanto, não estão conectadas entre si. Para que o circuito esteja fechado, deve-se conectar uma fileira com a outra usando, para isso, componentes eletrônicos. De um lado do circuito se deve ter uma entrada de energia, e do outro lado uma saída para o pino GND. Observe a figura seguinte:

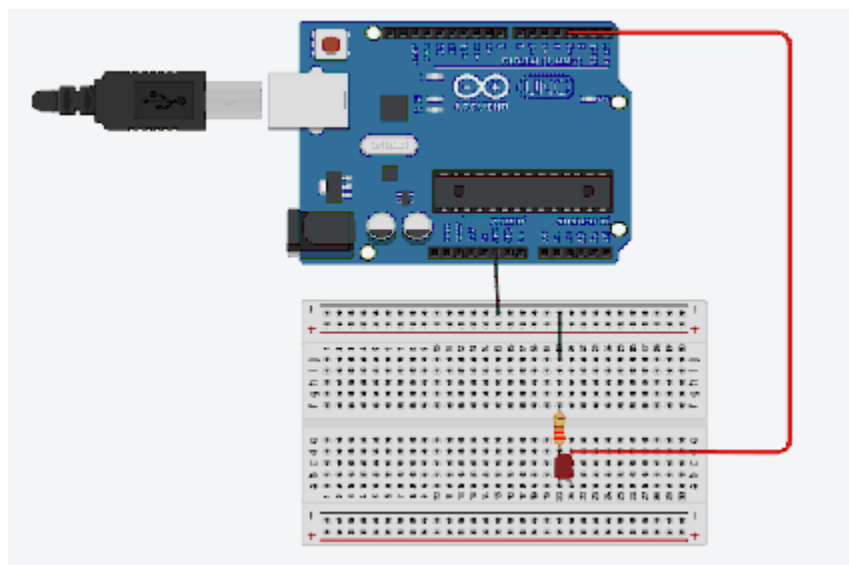


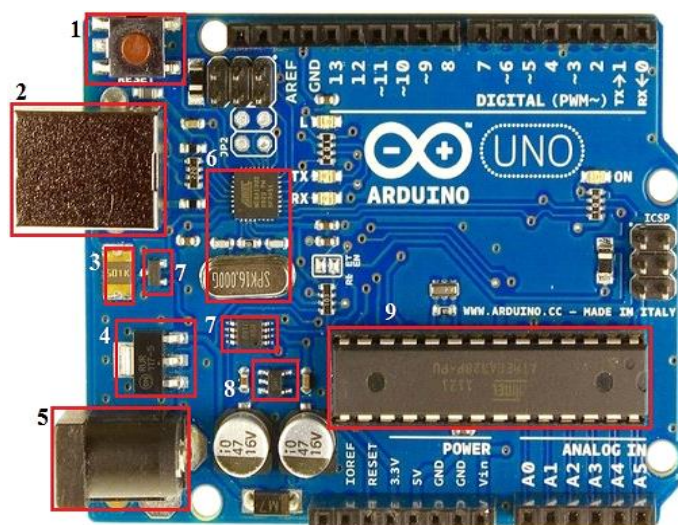
Figura 24: Esquema de circuito com Arduino, contendo um resistor e um LED.

Fonte: O Autor (2019).

Neste exemplo, o pino 3 do Arduino está ligado, através de um cabo jumper, a uma fileira da protoboard, onde há um LED vermelho. Se o Arduino for programado para enviar um sinal lógico alto (5 V) no pino 3, a fileira contendo o LED ficará com a mesma tensão também, acendendo o LED. Como o LED trabalha com 1,8 V, foi colocado um resistor de 220 Ω logo após o LED, para evitar que os 5 V o queime. O trajeto da corrente elétrica começa, então, no pino 3, segue para o LED, passa pelo resistor e energiza outra fileira da protoboard, que contém um cabo jumper preto, que leva a corrente para a fileira lateral externa da protoboard, que por sua vez está ligada ao pino GND, fechando o circuito.

3.7 ENTENDENDO O ARDUINO

A placa Arduino possui muitos componentes eletrônicos explicados na Figura 25. O micro controlador, propriamente dito, é o componente retangular, comprido e preto, na parte mais de baixo da placa. A entrada USB localiza-se no canto superior esquerdo.



- 1 - Botão de reset;
- 2 - Entrada USB;
- 3 - Protege a USB do computador contra correntes acima de 500mA;
- 4 - Regula tensão para 5V (DC);
- 5 - Conector DC;
- 6 - Microcontrolador e cristal (intermediários entre a USB e o computador);
- 7 - Verifica se há tensão DC. Caso não tenha, alimenta o Arduino pela entrada USB;
- 8 - Regula a tensão para 3,3V (DC);
- 9 - Microcontrolador do Arduino;

Figura 25: Arduino UNO, com alguns de seus componentes enumerados e explicados.

Fonte: O Autor (2019)

Capítulo 4

LINGUAGEM DE PROGRAMAÇÃO PARA ARDUINO

4.1 AMBIENTE DE DESENVOLVIMENTO INTEGRADO (IDE)

Para escrever um *sketch* e carregar o micro controlador Arduino é necessário ter o *software* Arduino IDE (*Integrated Development Environment*), que pode ser baixado no próprio site do Arduino, no link <https://www.arduino.cc/en/main/software>.

A interface do programa pode ser vista na Figura 26.

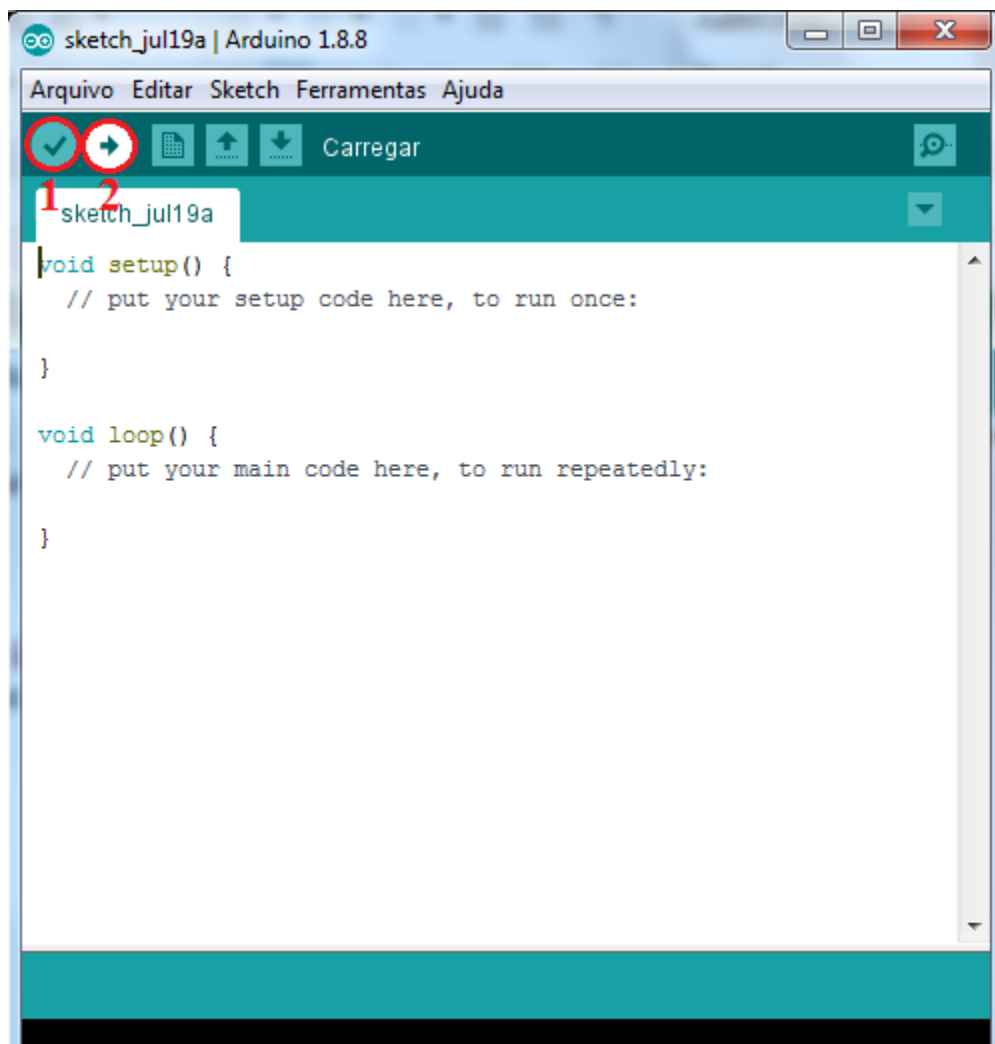


Figura 26: Interface do Arduino IDE.

Fonte: O Autor (2019).

No botão destacado em vermelho, com o número 1, se tem o comando "verificar", que checa o *sketch* e retorna ao programador se há algum erro na sintaxe. O botão com o número 2 é o "carregar", e é usado para carregar, na placa Arduino, o *sketch*. Para fazê-lo, no entanto, é necessário ter salvo o *sketch* previamente, bastando clicar em Arquivo -> Salvar Como.

Os outros três botões correspondem, da esquerda para a direita, aos comandos "Novo", "Abrir" e "Salvar".

É importante ressaltar que não será possível carregar o programa no microcontrolador se este não estiver conectado a uma entrada USB e se esta entrada não estiver discriminada no IDE. Assim, deve-se clicar em Ferramentas e então colocar o cursor em cima da opção Porta, selecionando a entrada USB em que o Arduino está conectado. Geralmente, no sistema operacional Windows, o nome das portas são COM. Caso haja mais de um Arduino ligado ao computador, e sejam modelos diferentes (existem vários: UNO, DUO, Duemilanove, Leonardo, entre outros), deve-se configurar o tipo de microcontrolador que está sendo usado, na opção Placa, acima de Porta.

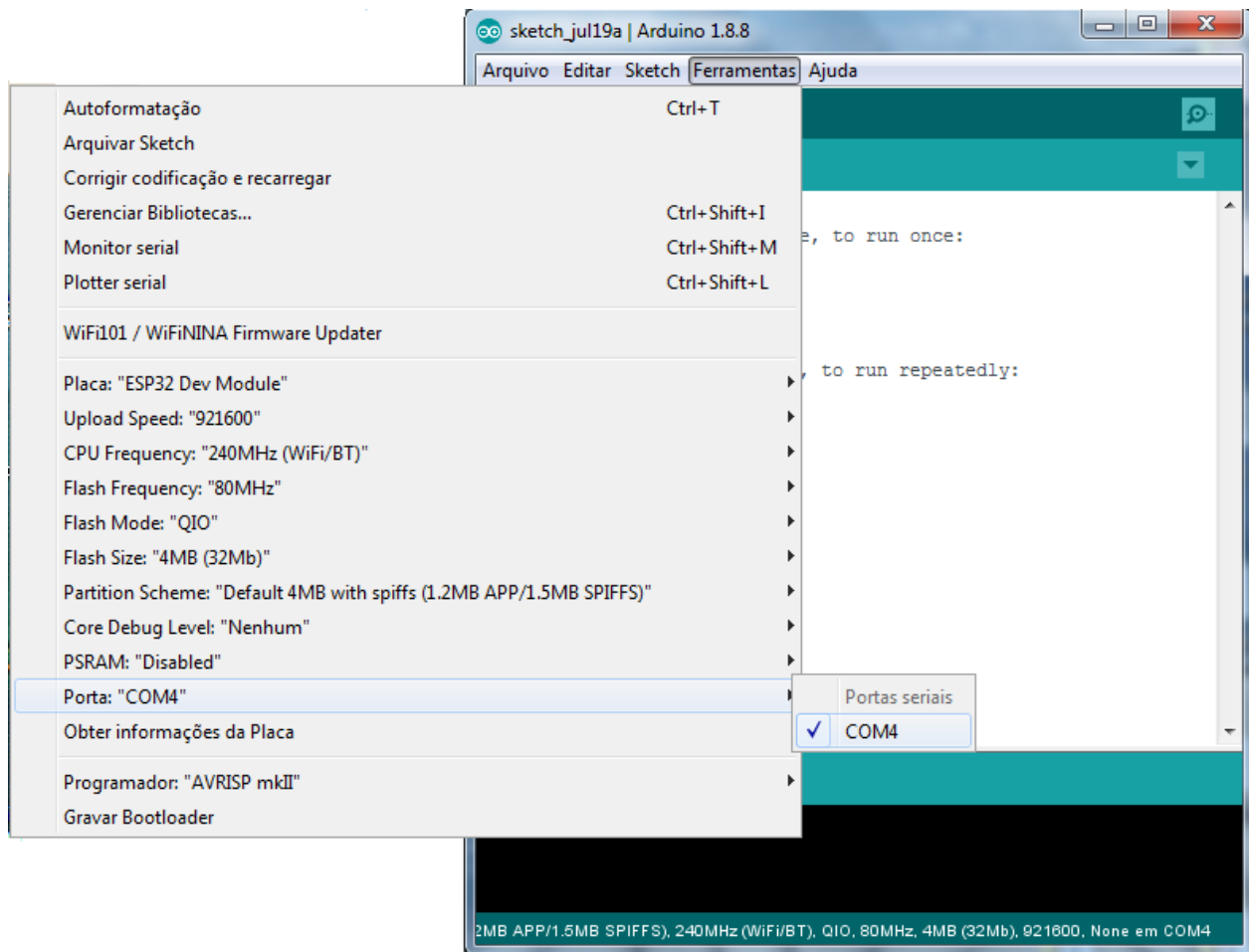


Figura 27: Selecionando a porta em que o Arduino está conectado, no Arduino IDE.

Fonte: O Autor (2019)

Feito o *download* da IDE, identificado o micro controlador e sua porta, tudo está pronto para se começar a criar projetos com o Arduino.

4.2 ESTRUTURA BÁSICA DO *SKETCH*

Os *sketchs* no Arduino possuem duas funções que praticamente sempre aparecerão: o void `setup()` e o void `loop()`.

A primeira função é usada para discriminar ações que serão executadas uma única vez pelo microprocessador. Geralmente, se coloca no void `setup()` as definições de INPUT e OUTPUT (entrada e saída de dados).

A segunda função apresentada é usada para discriminar ações que serão executadas inúmeras vezes, enquanto o microprocessador estiver ligado a uma fonte de energia. Por isso o nome da função é *loop*, visto que os comandos ficam se repetindo. Geralmente, a maior parte do *sketch* se encontra dentro do `void loop()`.

Para ligar o LED, como o circuito na imagem 24, segue-se os seguintes passos:

1º - declara-se o pino em que o LED está ligado como OUTPUT;

2º - atribui-se nível lógico alto (HIGH) para este pino.

O *sketch* ficará da seguinte forma:

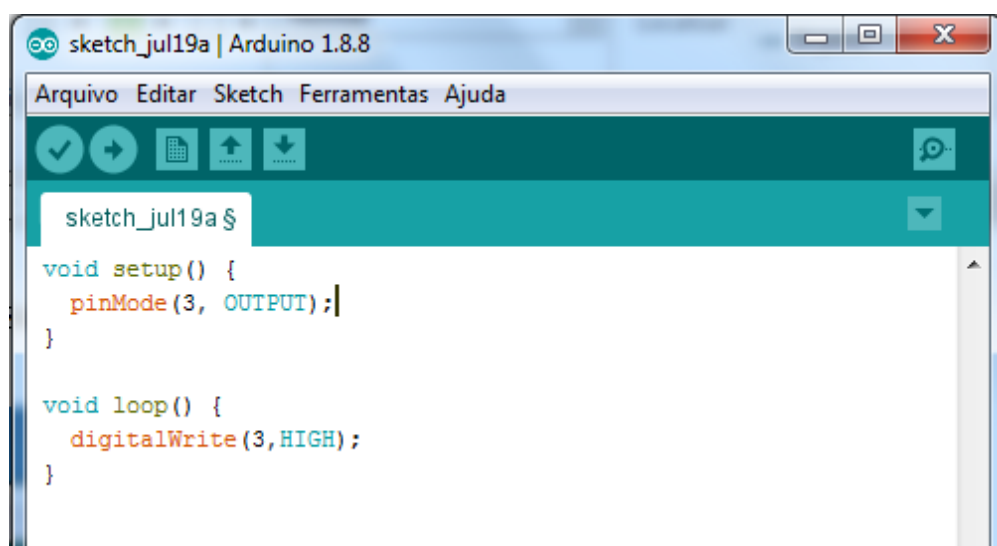


Figura 28: Sketch que acende um LED conectado ao pino 3 do Arduino.

Fonte: O Autor (2019)

O primeiro passo é feito uma única vez, logo se encontra dentro da função `void setup()`. O segundo passo continuará sendo executado, logo se encontra dentro da função `void loop()`.

O comando `pinMode(3, OUTPUT);` serve para informar o Arduino que o pino 3 estará funcionando como OUTPUT, ou seja, estará enviando informações ao usuário. Para configurar o pino 6 também como INPUT, deve-se escrever `pinMode(6, INPUT);` e assim sucessivamente para qualquer pino, seja OUTPUT ou INPUT.

O comando `digitalWrite(3, HIGH);` serve para informar o Arduino que um nível lógico alto (5 V) deve ser enviado ao pino 3. Ao fazer isso, o LED que está associado a esse pino irá acender.

Note que tanto `void setup(){` como `void loop(){` começam abrindo chaves { e terminam fechando as chaves }. Tudo o que se encontra dentro das chaves está dentro dessas funções.

4.3 ACENDENDO E APAGANDO UM LED

Com as informações dadas até este ponto, pode-se acender e apagar um LED, usando o mesmo circuito apresentado na Figura 24. O *sketch* segue o seguinte padrão: primeiramente, se diz que o pino em que o LED está conectado é OUTPUT; depois informa-se o microcontrolador que se quer ligar o LED; seguido então de um comando para o apagar. O Arduino executará esses dois comandos de forma muito rápida, parecendo que o LED está aceso permanentemente. Assim, se faz necessário especificar o tempo que o LED ficará aceso ou apagado, utilizando o comando `delay()`. Observe o *sketch*:



```
sketch_jul19a | Arduino 1.8.8
Arquivo Editar Sketch Ferramentas Ajuda
sketch_jul19a $
void setup() {
  pinMode(3, OUTPUT);
}

void loop() {
  digitalWrite(3, HIGH);
  delay(1000);
  digitalWrite(3, LOW);
  delay(1000);
}
```

Figura 29: Sketch que faz um LED piscar de um em um segundo.

Fonte: O Autor (2019)

O comando `delay(1000)` indica quanto tempo o microcontrolador deve esperar após executar um comando. O número, dentro dos parênteses, representa a quantidade em milissegundos que se esperará. Assim, `delay(1000)` significa que o Arduino ficará 1000 milissegundos, ou 1 segundo, sem executar os comandos seguintes. Ao final da função `void loop()`, indicado pelas chaves `}`, tudo que está dentro da função se repete, ou seja, o *sketch* apresentado faz com que o LED ligue por 1 segundo e apague pelo mesmo tempo, repetidas vezes.

4.4 HIGH/LOW, 0/1, TRUE/FALSE

Os comandos digitais do Arduino, tais como o `digitalwrite`, reconhecem dois valores somente, onde um possui nível lógico alto e outro nível lógico baixo. Quando se escreve o comando `digitalwrite(3, HIGH)`; se está colocando o pino 3 do Arduino em nível lógico alto, ou seja, uma tensão de 5 V passará pelo pino e pelo componente eletrônico que estiver conectado ao pino. O oposto é o comando `digitalwrite(3, LOW)`; que coloca o nível lógico baixo no pino 3 (0 V estariam passando).

Existe mais de uma forma de atribuir nível lógico alto ou baixo através de comandos digitais. Se for escrito, no lugar de `HIGH`, o valor *true*, ou até mesmo 1, o resultado será o mesmo. Para nível lógico baixo, pode-se escrever, além de `LOW`, *false* e 0. Assim, temos que:

```
digitalwrite(3, HIGH); = digitalwrite(3, true); = digitalwrite(3, 1);  
digitalwrite(3, LOW); = digitalwrite(3, false); = digitalwrite(3, 0);
```

4.5 VARIÁVEIS NA PROGRAMAÇÃO DO ARDUINO

No capítulo 2 foi visto que as variáveis são espaços na memória reservados para se guardar algum valor, e que existem tipos de variáveis que guardam valores específicos. As classes de variáveis apresentadas no português estruturado foram *inteiro*, *real* e *caractere*, onde a primeira guarda valores numéricos inteiros, a segunda guarda valores numéricos decimais ("número quebrado") e a terceira guarda letras, nomes e frases.

Na linguagem do Arduino, se tem maior diversidade de variáveis. Os principais tipos são as numéricas, as variáveis de texto e as variáveis booleanas, explicadas a seguir.

- Variáveis numéricas: São variáveis que armazenam números inteiros ou números reais.

a) *Números inteiros*: Podem ser dos seguintes tipos:

- I. byte: guarda valores de 0 a 255.
- II. int: guarda valores de -32.758 a 32.758.
- III. unsigned int: guarda valores de 0 a 65.535.
- IV. long: guarda valores de -2.147.483.648 a 2.147.483.648.
- V. unsigned long: guarda valores de 0 a 4.294.967.295.

b) *Números reais*: Podem ser dos seguintes tipos:

- I. float: guarda valores de -3,402e38 a 3,402e38.
- II. double: Costuma ser igual ao float, mas em alguns Arduinos tem alcance maior.

- Variáveis de texto: Podem ser as seguintes:

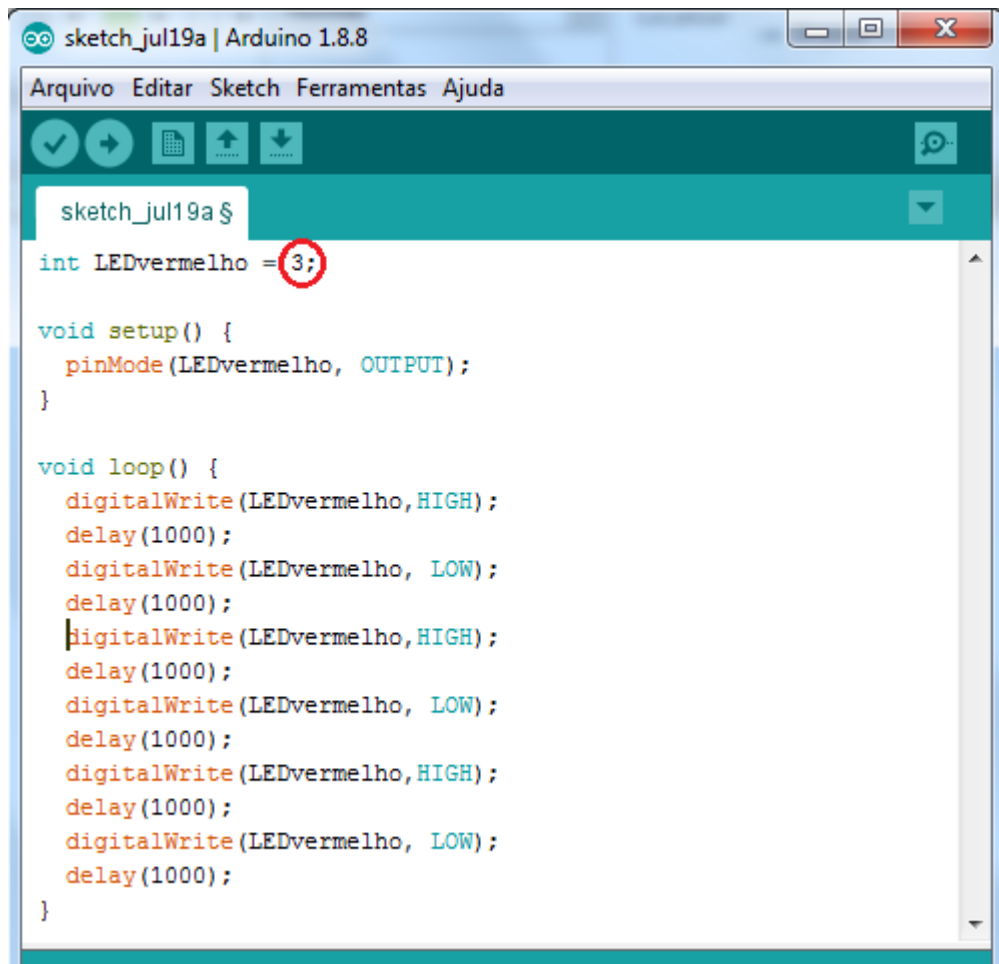
- I. char : aceita um caractere, armazenando-o em uma escala de -128 a 127.
- II. string : aceita texto.

- Variáveis booleanas: Este tipo de variável aceita somente dois valores: 1 ou 0, que é o mesmo que HIGH ou LOW, ou ainda *true* ou *false*. É declarada como *bool* ou *boolean*.

4.6 VARIÁVEIS PÚBLICAS E PRIVADAS

Ao criar um *sketch*, convém que sejam declaradas, antes das funções `setup` e `loop`, algumas variáveis que serão usadas ao longo do programa. Por exemplo, se um LED vermelho conectado ao pino 3 é ligado e desligado diversas vezes ao longo do programa, pode-se declarar uma variável chamada `LEDvermelho` e atribuir a ela o valor 3. Assim, sempre que se quiser ligar/desligar o LED, basta usar o comando `digitalWrite(LEDvermelho, HIGH)` ou `digitalWrite(LEDvermelho,`

LOW). Para mudar o pino ao qual o LED está conectado, só é preciso mudar o *sketch* em um ponto (onde foi declarada a variável). Caso haja variável declarada, será necessário mudar todos os comandos onde o pino 3 apareça, colocando o número do novo pino. Observe a Figura 30.



```
sketch_jul19a $
int LEDvermelho = 3;

void setup() {
  pinMode(LEDvermelho, OUTPUT);
}

void loop() {
  digitalWrite(LEDvermelho, HIGH);
  delay(1000);
  digitalWrite(LEDvermelho, LOW);
  delay(1000);
  digitalWrite(LEDvermelho, HIGH);
  delay(1000);
  digitalWrite(LEDvermelho, LOW);
  delay(1000);
  digitalWrite(LEDvermelho, HIGH);
  delay(1000);
  digitalWrite(LEDvermelho, LOW);
  delay(1000);
}
```

Figura 30: Sketch que declara uma variável pública e faz um LED piscar.

Fonte: O Autor (2019)

Como pode ser visto na figura acima, para mudar o pino onde o LED se encontra, deve-se mudar somente o valor destacado no círculo vermelho. Observe agora como será mais trabalhoso mudar o pino caso o *sketch* esteja escrito sem declarar a variável LEDvermelho:

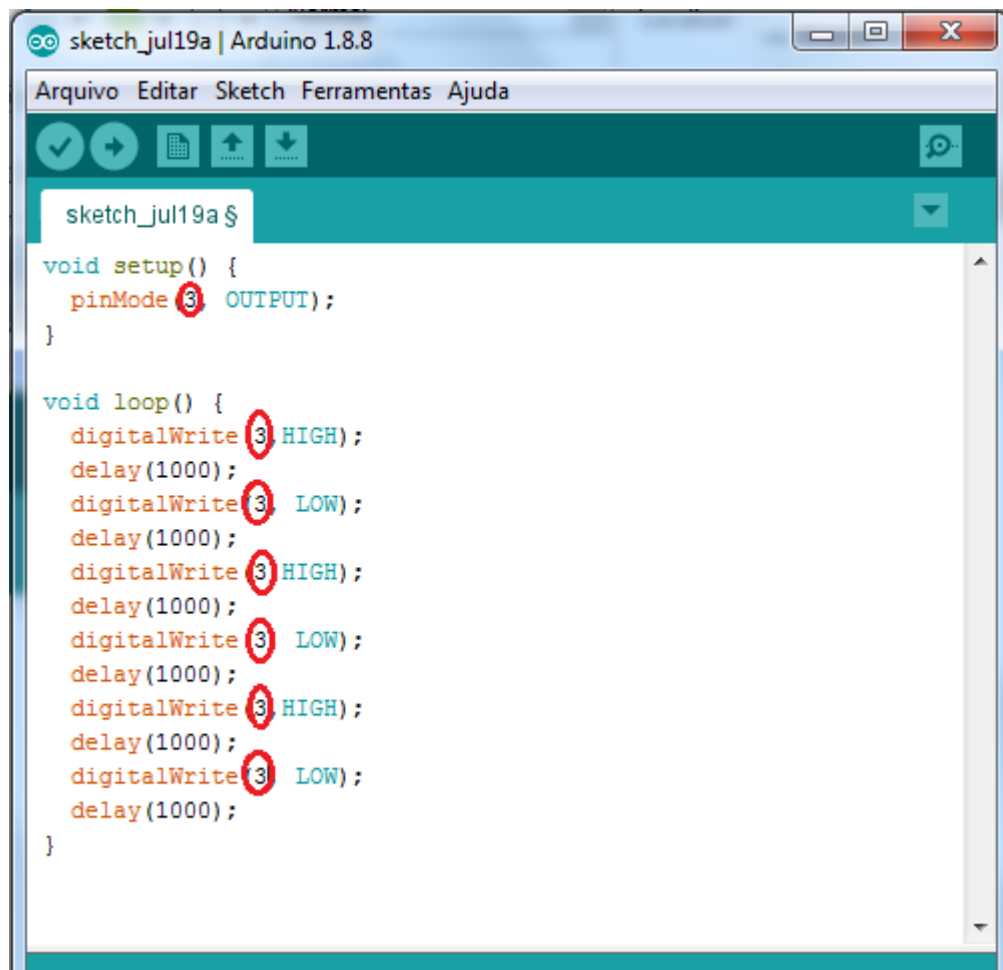


Figura 31: Sketch sem variável pública que faz um LED piscar.

Fonte: O Autor (2019)

Será necessário, neste caso, mudar todos os números destacados com um círculo vermelho, caso se quisesse mudar o pino do LED.

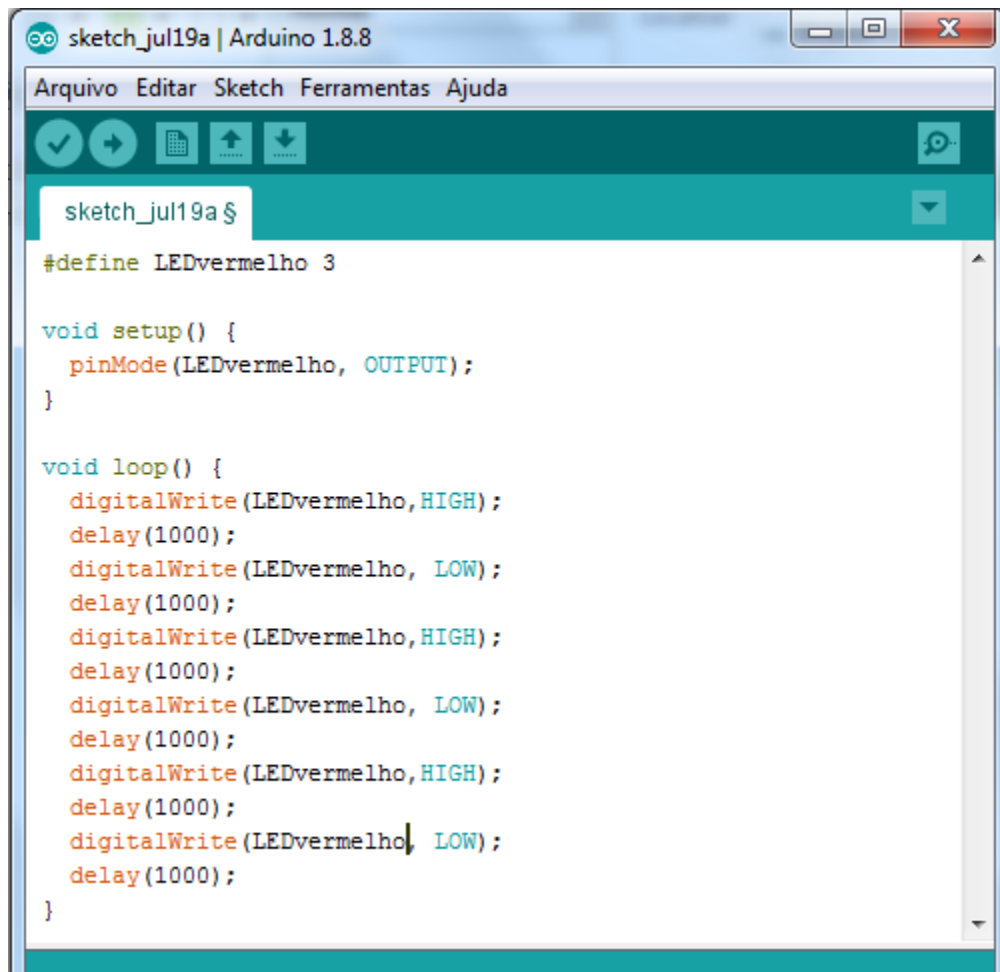
Ao criar variáveis antes da função `setup` e da função `loop`, estas ficarão disponíveis o tempo todo para o programa, sendo chamadas de **públicas**, já que foram criadas fora das funções e podem ser usadas a qualquer hora, em qualquer lugar, no *sketch* do programa.

Se forem criadas variáveis dentro da função `setup` ou da função `loop`, serão variáveis **privadas**, que só poderão ser usadas pela função em que se encontram.

4.7 #DEFINE

Para tornar o *sketch* mais leve, as variáveis declaradas que representam os pinos do Arduino podem ser definidas usando o comando `#define`, fazendo com que não ocupem espaço na memória do Arduino, e na prática o efeito será o mesmo.

Assim, para fazer o LED piscar algumas vezes, demorando 1 segundo aceso e 1 segundo apagado, pode-se usar (também) o *sketch* abaixo:

The image shows a screenshot of the Arduino IDE interface. The title bar reads 'sketch_jul19a | Arduino 1.8.8'. The menu bar includes 'Arquivo', 'Editar', 'Sketch', 'Ferramentas', and 'Ajuda'. Below the menu is a toolbar with icons for saving, running, and other functions. The main text area contains the following C++ code:

```
#define LEDvermelho 3

void setup() {
  pinMode(LEDvermelho, OUTPUT);
}

void loop() {
  digitalWrite(LEDvermelho, HIGH);
  delay(1000);
  digitalWrite(LEDvermelho, LOW);
  delay(1000);
  digitalWrite(LEDvermelho, HIGH);
  delay(1000);
  digitalWrite(LEDvermelho, LOW);
  delay(1000);
  digitalWrite(LEDvermelho, HIGH);
  delay(1000);
  digitalWrite(LEDvermelho, LOW);
  delay(1000);
}
```

Figura 32: Sketch que faz o LED piscar, usando o comando `#define`.

Fonte: O Autor (2019).

4.8 O MONITOR SERIAL

O monitor serial do Arduino está presente no *software* Arduino IDE. É uma tela usada para enviar informações, do Arduino, para o usuário, podendo ser acessada no botão destacado em vermelho na figura abaixo:

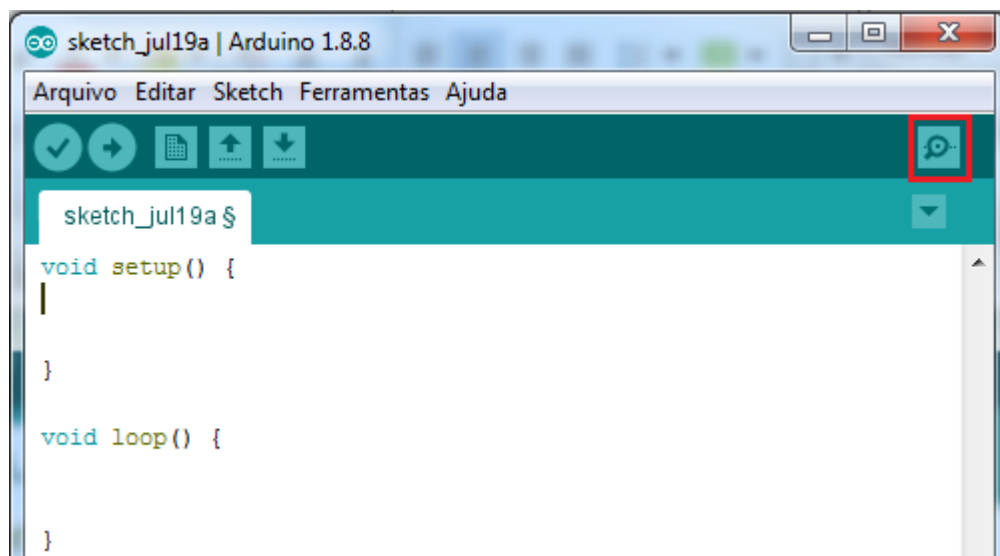


Figura 33: Arduino IDE com botão do monitor serial destacado.

Fonte: O Autor (2019)

Para mostrar informações no monitor serial usa-se os comandos `Serial.print` e `Serial.println`, onde o primeiro só mostra uma mensagem e o último mostra uma mensagem e pula uma linha.

Para usar os comandos acima, no entanto, deve-se, primeiramente, definir a velocidade com que o Arduino enviará informações para o Monitor Serial. Para saber a velocidade correta (o que pode ser configurado), abre-se o Monitor Serial e, no canto inferior esquerdo, se procura a velocidade, como na figura abaixo, marcada em amarelo:

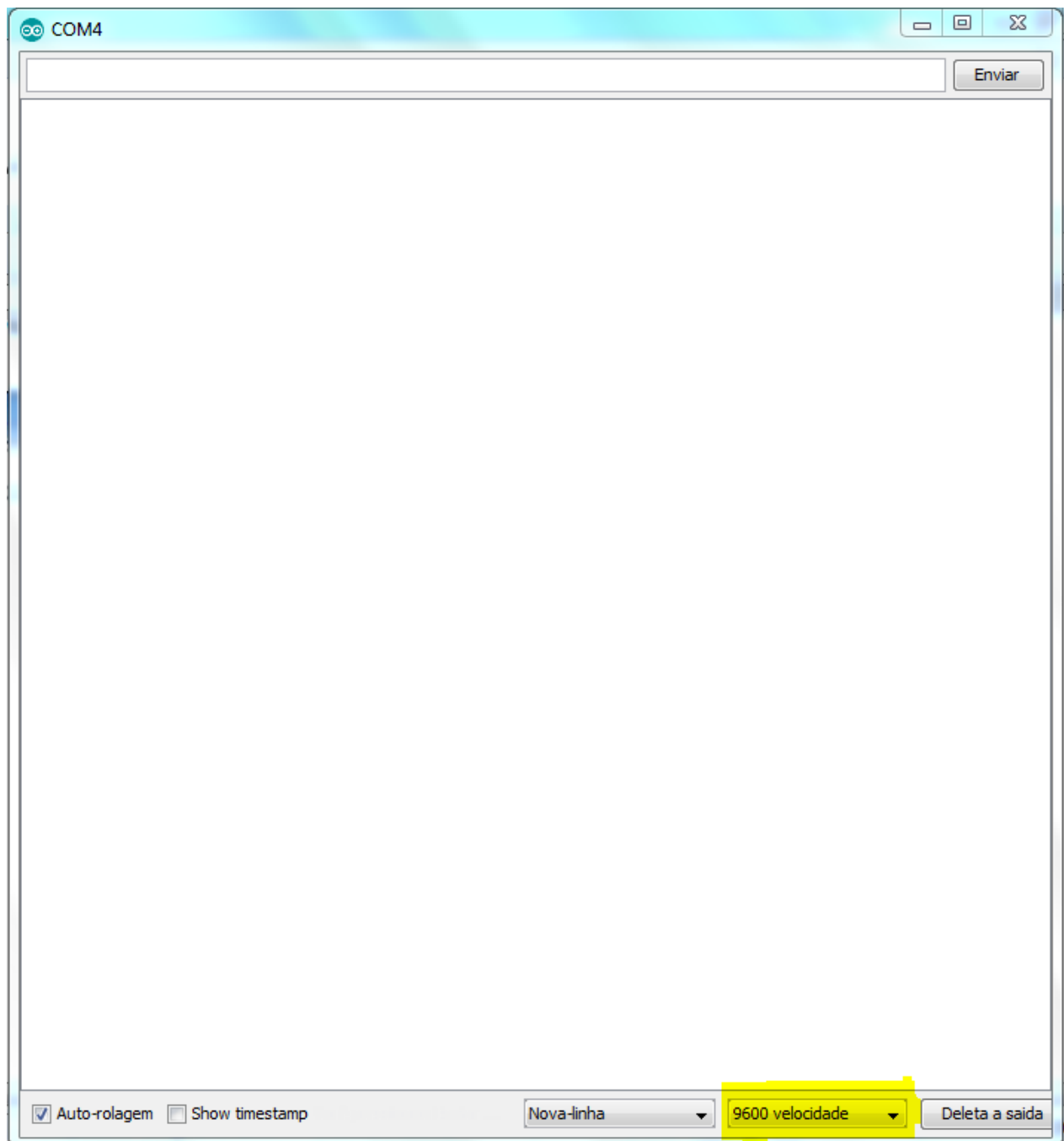
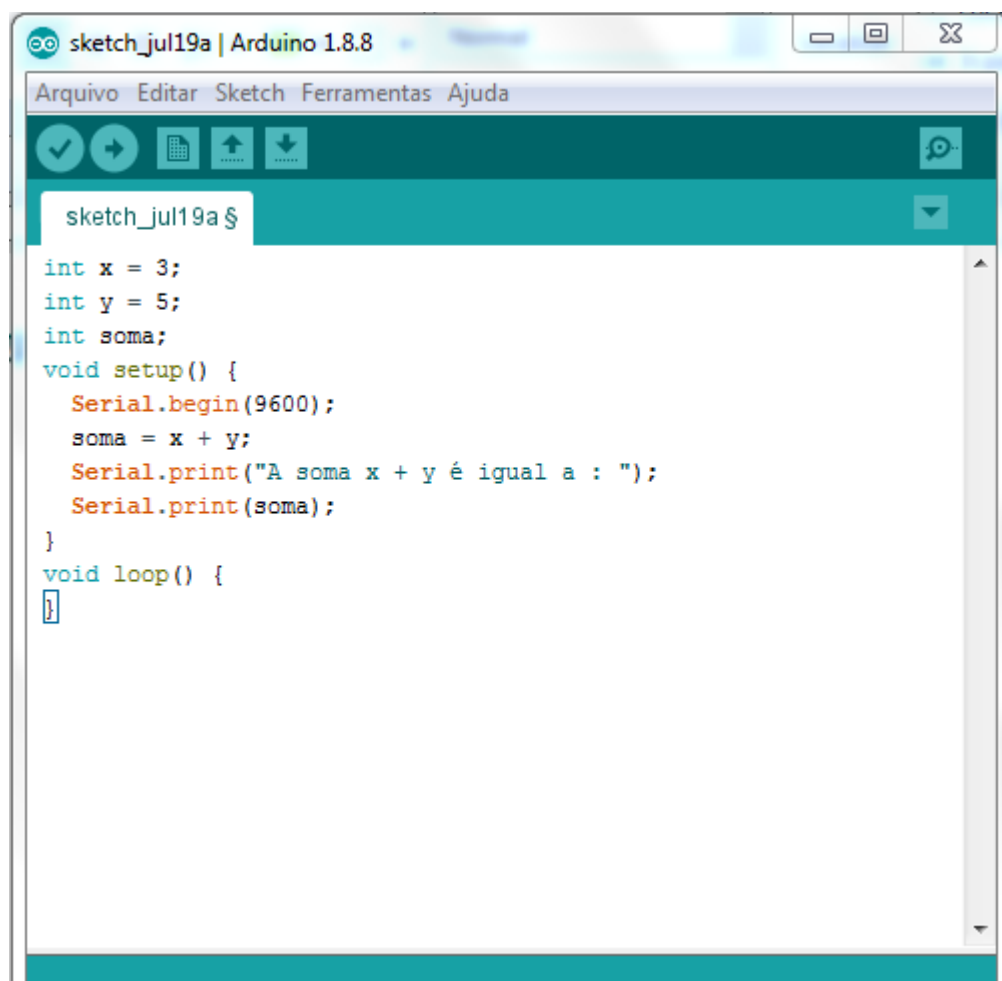


Figura 34: Monitor serial, com a velocidade destacada.

Fonte: O Autor (2019)

Por padrão, a velocidade é 9600 bytes por segundo. Assim, no *sketch* deve constar, dentro da função `setup`, o comando `Serial.begin(9600);` que inicializa o Monitor Serial e o prepara para ser usado.

Um exemplo: para se escrever um *sketch* que soma dois números (3 e 5, por exemplo) e mostra o resultado no Monitor Serial, deve-se prosseguir da seguinte forma:



```
sketch_jul19a | Arduino 1.8.8
Arquivo Editar Sketch Ferramentas Ajuda
sketch_jul19a $
int x = 3;
int y = 5;
int soma;
void setup() {
  Serial.begin(9600);
  soma = x + y;
  Serial.print("A soma x + y é igual a : ");
  Serial.print(soma);
}
void loop() {
}
```

Figura 35: Sketch que executa uma soma e mostra a conta no Monitor serial.

Fonte: O Autor (2019)

Ao carregar uma placa Arduino com esse *sketch*, aparecerá a seguinte mensagem “A soma de $x + y$ é igual a 8”, no Monitor Serial, conforme a Figura 36.

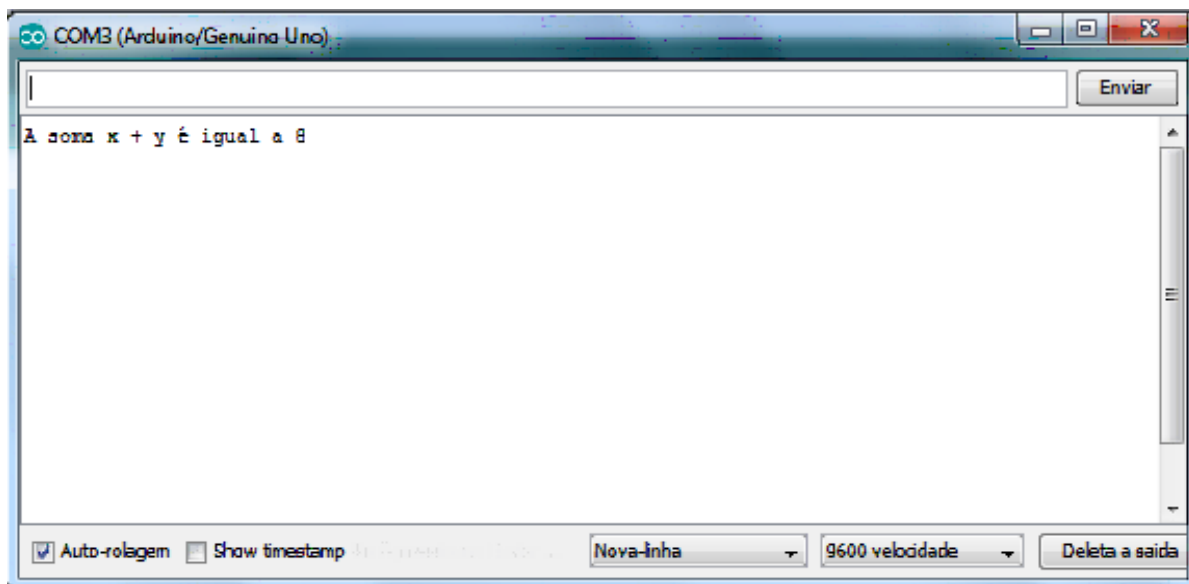


Figura 36: Monitor serial após rodar o sketch da Figura 35.

Fonte: O Autor (2019)

A função *loop* do *sketch* está vazia, e todas as informações estão dentro da função *setup*. Isso ocorre porque não se quer executar a conta inúmeras vezes, mas uma única vez. Deve-se sempre ter as características das duas funções em mente: uma serve para comandos que serão executados uma única vez, enquanto que a outra serve para comandos que serão executados repetidas vezes.

É importante sempre atentar, também, à velocidade declarada no *sketch*, com o comando `Serial.begin`, que deve ser igual à do Monitor Serial. Se as velocidades forem diferentes, o programa não funcionará corretamente.

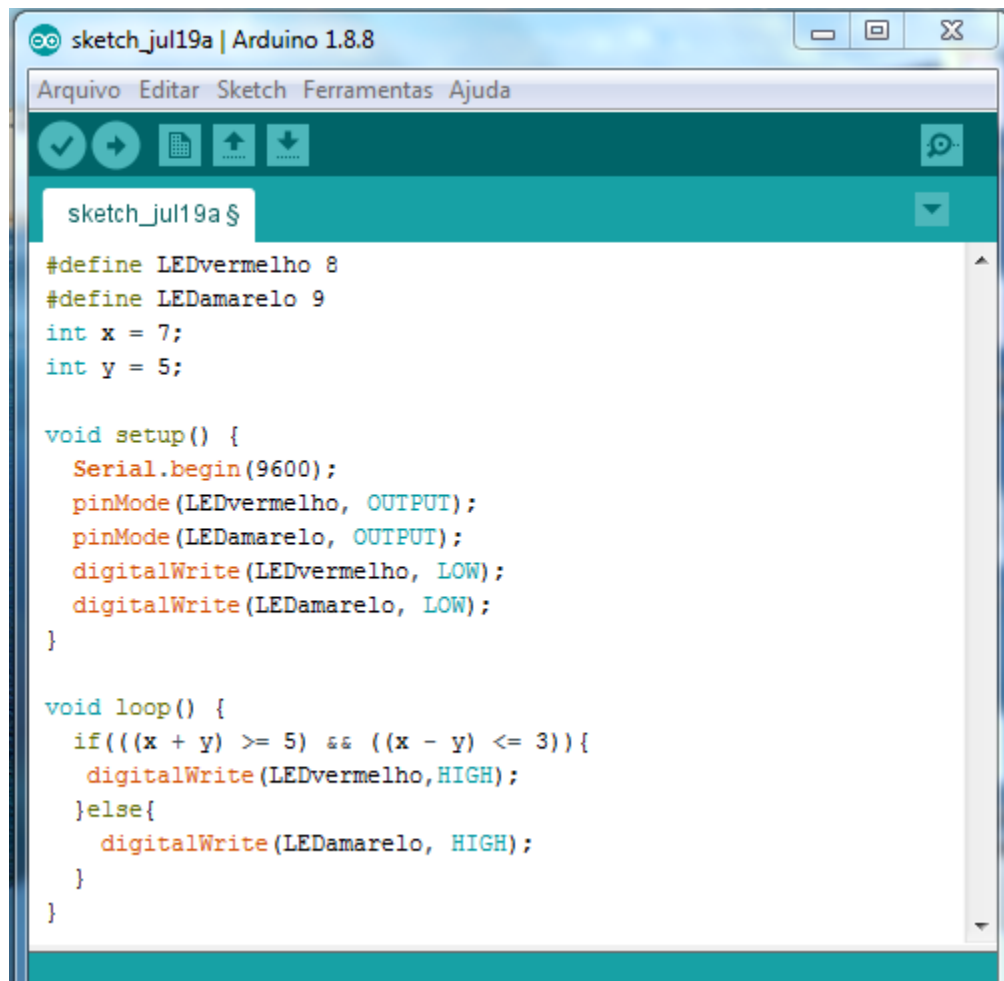
4.9 OPERADORES, CONDICIONAIS

Assim como explicado no Capítulo 2, sempre que se quer somar, subtrair, ou executar outras operações matemáticas, utiliza-se os operadores: `+`, `-`, `/`, `%`, `*`, entre outros. Estes são os operadores matemáticos, e são os mesmos usados na linguagem do Arduino.

Se precisar comparar dois valores, pode-se usar os operadores relacionais, como `>`, `<`, `>=`, `<=`, `==` e `!=`, também apresentados no Capítulo 2.

Se precisar satisfazer duas ou mais condições, pode-se usar os operadores lógicos `&&` e `||`. Unindo operadores com funções condicionais, é possível criar uma grande variedade de *sketches*.

As condicionais, no português estruturado, são representadas pelo comando "se". No Arduino IDE, esse comando será o *if*. O comando "senao" do português estruturado será o *else*. A lógica do comando é igual à apresentada no Capítulo 2, conforme o *sketch* da Figura 37.

The image shows a screenshot of the Arduino IDE interface. The title bar reads 'sketch_jul19a | Arduino 1.8.8'. The menu bar includes 'Arquivo', 'Editar', 'Sketch', 'Ferramentas', and 'Ajuda'. Below the menu bar is a toolbar with icons for checking, running, serial monitor, and other functions. The main text area contains the following C++ code:

```
sketch_jul19a $
#define LEDvermelho 8
#define LEDamarelo 9
int x = 7;
int y = 5;

void setup() {
  Serial.begin(9600);
  pinMode(LEDvermelho, OUTPUT);
  pinMode(LEDamarelo, OUTPUT);
  digitalWrite(LEDvermelho, LOW);
  digitalWrite(LEDamarelo, LOW);
}

void loop() {
  if(((x + y) >= 5) && ((x - y) <= 3)){
    digitalWrite(LEDvermelho, HIGH);
  }else{
    digitalWrite(LEDamarelo, HIGH);
  }
}
```

Figura 37: Sketch que usa a função condicional *if*, seguido de *else*.

Fonte: O Autor (2019)

O programa acima possui uma função condicional *if*. Para que o comando `digitalWrite(LEDvermelho, HIGH);` seja executado, duas condições precisam ser satisfeitas. A primeira delas é que a soma de *x* e *y* seja maior ou igual a 5 e a segunda condição é que a diferença entre *x* e *y* seja menor que 3. Se ambas condições forem satisfeitas, o comando será executado. Se

não forem satisfeitas, o comando `digitalWrite(LEDamarelo, HIGH)`; que está dentro da função *else*, será executado.

4.10 FUNÇÕES DE REPETIÇÃO

No português estruturado, se tem como funções de repetição os comandos "para" e "enquanto". Na linguagem do Arduino, os comandos são o `for` e o `while`, respectivamente. A estrutura não será modificada.

Assim, para escrever um *sketch* que acenda um LED e o apague repetidas vezes, e depois passe para outro LED, se deve proceder da seguinte maneira:

The image shows the Arduino IDE interface with a sketch named 'sketch_sep09a'. The code defines two LEDs, 'LEDvermelho' (red) and 'LEDamarelo' (yellow), at pins 8 and 9 respectively. The 'setup' function initializes both pins as outputs and sets them to LOW. The 'loop' function contains two 'while' loops. The first loop blinks the red LED by setting it HIGH for 300ms and then LOW for 300ms, repeating this 10 times. The second loop does the same for the yellow LED. A counter variable 'c' is used to control the repetition of each loop.

```
sketch_sep09a $
#define LEDvermelho 8
#define LEDamarelo 9

void setup() {

  pinMode(LEDvermelho, OUTPUT);
  pinMode(LEDamarelo, OUTPUT);
  digitalWrite(LEDvermelho, LOW);
  digitalWrite(LEDamarelo, LOW);
}

void loop() {
  int c = 1;
  while(c <= 10){
    digitalWrite(LEDvermelho, HIGH);
    delay(300);
    digitalWrite(LEDvermelho, LOW);
    delay(300);
    c = c + 1;
  }
  c = 1;
  while(c <= 10){
    digitalWrite(LEDamarelo, HIGH);
    delay(300);
    digitalWrite(LEDamarelo, LOW);
    delay(300);
    c += 1;
  }
}
```

Figura 38: Sketch que usa o comando de repetição while para fazer um LED piscar.

Fonte: O Autor (2019)

O comando for também pode ser usado para escrever o *sketch* da Figura 38, da seguinte forma:

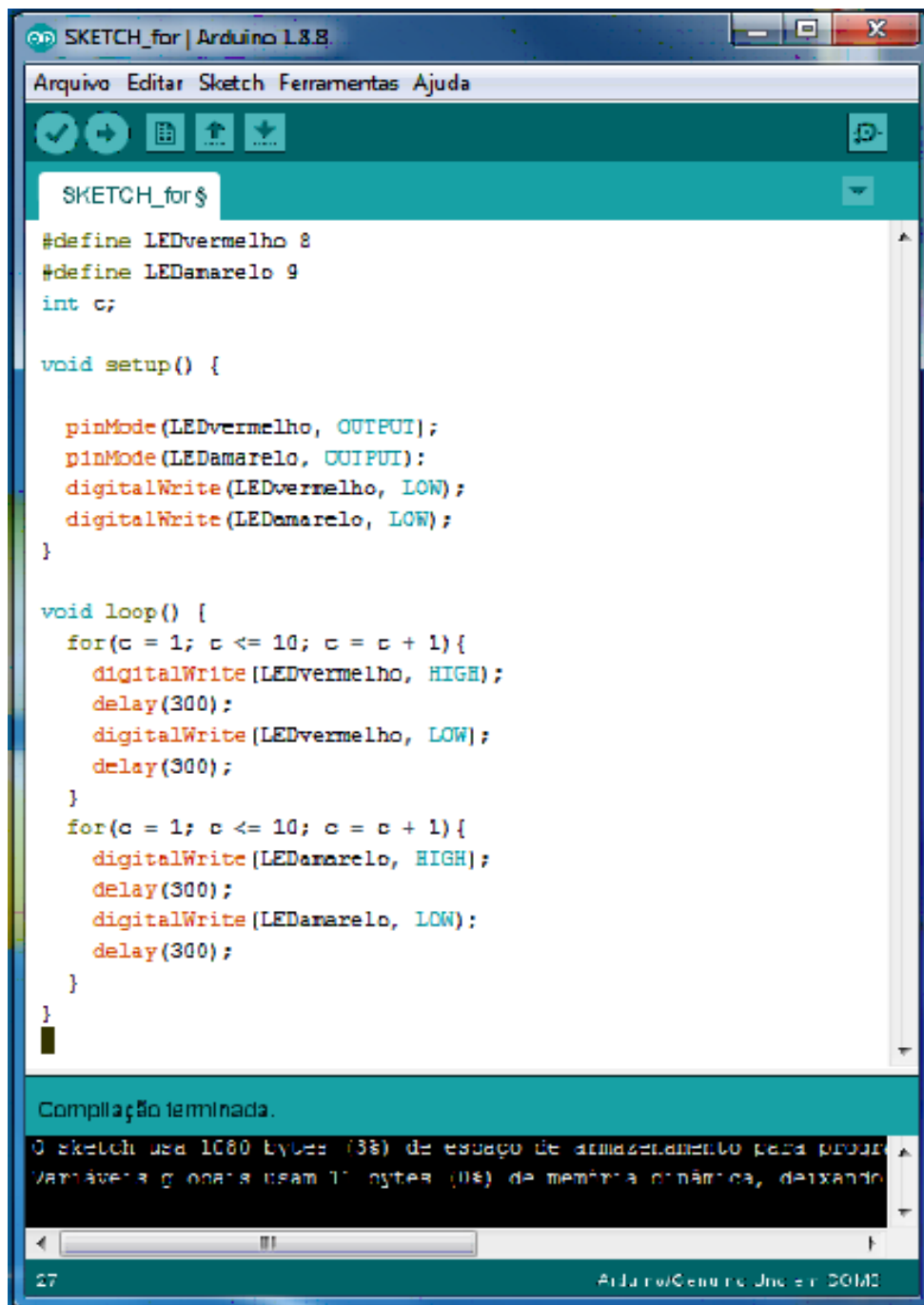


Figura 39: Sketch que usa o comando de repetição for para fazer LEDs piscarem.

Fonte: O Autor (2019)

Note que, nos exemplos dados acima, as funções de repetição não são imprescindíveis para o bom funcionamento do programa. Pode-se ordenar ao Arduino, por exemplo, que acenda e apague cada LED dez vezes, usando o comando `digitalWrite` e `delay`. Entretanto, em programas

mais extensos, fica muito mais leve e proveitoso usar funções de repetição. Por exemplo, escrever um código que executa uma sequência de comandos por cem ou mais vezes é muito cansativo, além de que o processamento dessa quantidade enorme de informação é pesado para o microprocessador Arduino. Neste caso, as funções `for` e `while` se adequam perfeitamente.

4.11 PINMODE

Já foi explicado o que são variáveis públicas e privadas, como usar o `#define`, como usar os diversos operadores e como empregar devidamente funções de repetição e condicionais.

No tópico 4.2 já foi citado o tema aqui abordado, mas deve ser frisado que, se for construído um circuito onde se tem um LED ligado ao pino 9 do Arduino, por exemplo, deve-se especificar no código que o pino 9 é um OUTPUT. Ou seja, deve-se especificar que o pino tem uma *saída*, de onde alguma informação irá sair para o usuário. No caso do LED, a informação que sairá será em forma de luz. Qualquer componente eletrônico de onde saia alguma informação ao usuário será configurado como OUTPUT.

Assim, de maneira diferente do LED, se configura um botão (*push button*) por exemplo. Neste caso, o botão será configurado como INPUT, visto que é um meio do usuário *inserir* informações, e não recebê-las. O mesmo pode ser dito de um sensor de temperatura ou luminosidade, já que os dados lidos pelos sensores são informações que serão dadas ao Arduino, e não recebidas.

As definições devem ser feitas dentro da função `setup`, já que serão executadas uma única vez.

4.12 USANDO UM PUSH BUTTON

O *push button*, como foi dito no tópico 4.11, deve ser declarado como INPUT dentro da função `setup`. Antes das funções `loop` e `setup`, deve ser especificado em qual pino o *push button* está ligado. Uma vez feito, é necessário entender como o componente funciona para montar o circuito de maneira adequada.

O *push button* possui dois pares de pinos, que serão encaixados na *protoboard*. Cada pino está ligado ao seu par, mas não aos outros dois pinos. Ao apertar o botão, uma ligação entre os dois pares de pinos é feita, permitindo que a energia transite. Assim, quando se coloca um botão no circuito, para que a energia passe, o botão deve ser pressionado. Se não o for, o circuito estará aberto.

Pode-se montar um circuito com *push button* da seguinte maneira: um fio *jumper* ligando um lado do *push button* ao Arduino, em um pino digital; outro fio *jumper* ligando o outro lado do *push button* ao pino GND (terra) do Arduino. Desta forma, usando o comando `digitalRead` é possível verificar se o botão está apertado ou não. Observe o esquema do circuito, na Figura 40.

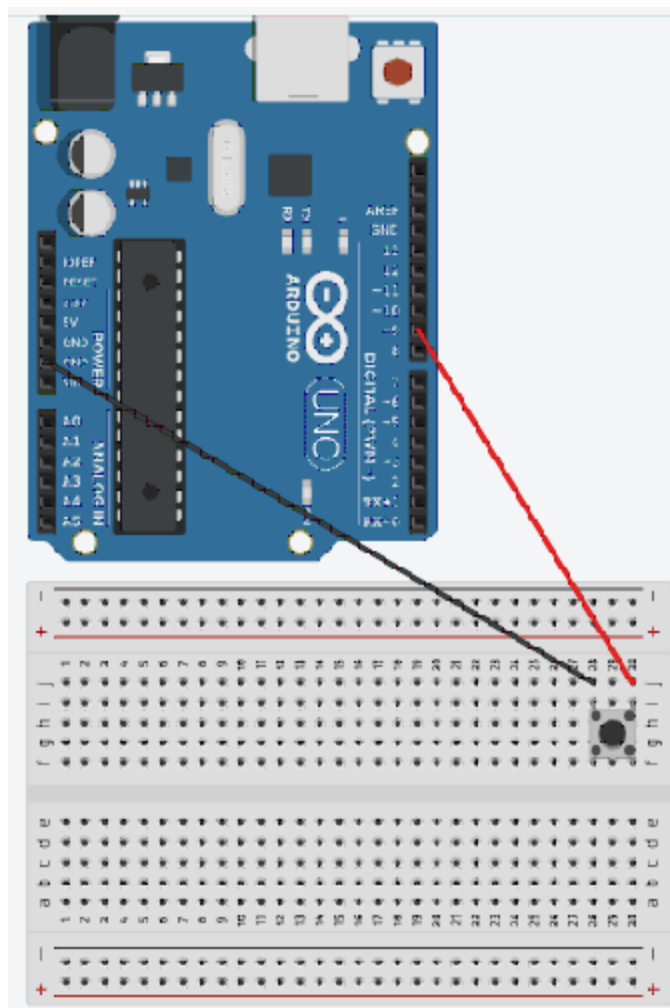


Figura 40: Circuito com um push button.

Fonte: O Autor (2019)

Na Figura 40, ao utilizar o comando `digitalRead(9)`; se está ordenando que o Arduino verifique se o botão, ligado ao pino 9, está apertado ou não. Pode-se colocar uma condicional, logo abaixo do comando, ordenando que o microprocessador faça algo caso o botão esteja sendo apertado. O *sketch* da Figura 41 mostra como se pode verificar se o botão está apertado e mostrar a informação no monitor serial.

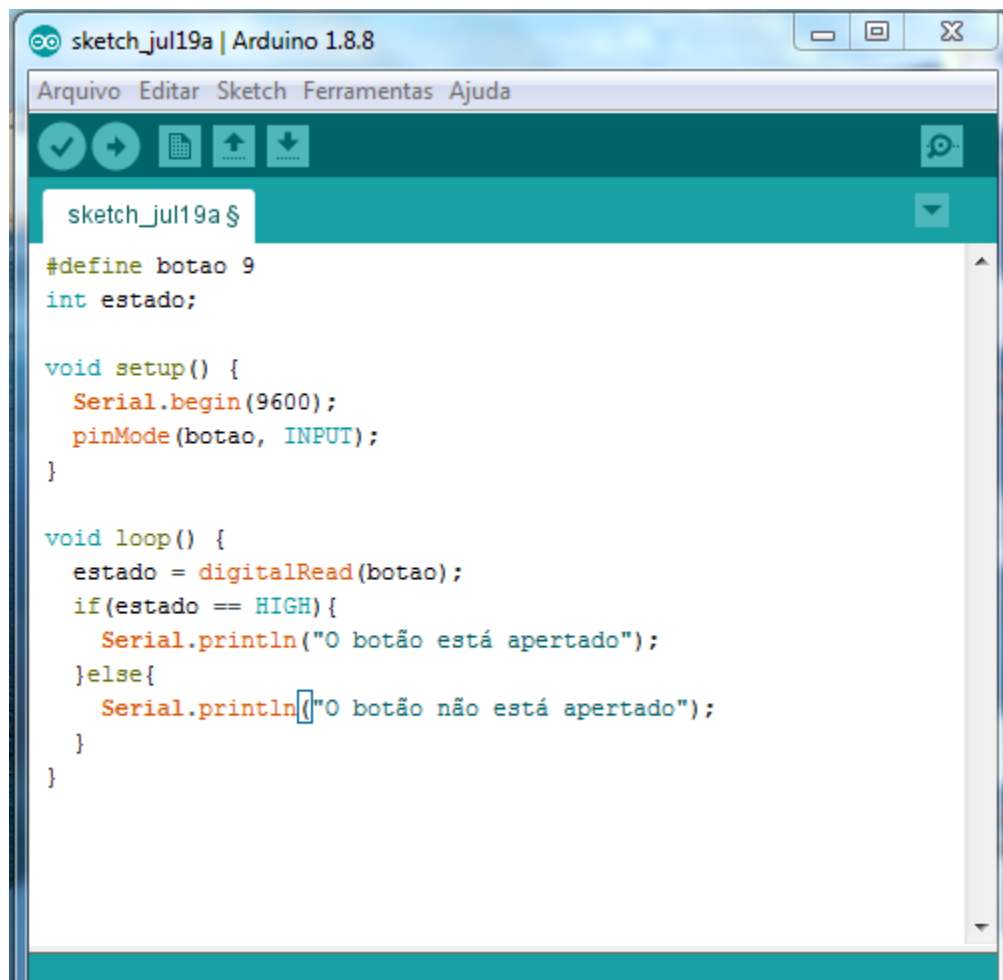
The image shows the Arduino IDE interface with a sketch named 'sketch_jul19a'. The code defines a pin constant 'botao' as 9 and an integer variable 'estado'. In the 'setup' function, it initializes the serial port at 9600 baud and configures pin 9 as an input. The 'loop' function reads the digital value of pin 9 into 'estado'. If 'estado' is HIGH, it prints 'O botão está apertado' to the serial monitor; otherwise, it prints 'O botão não está apertado'.

Figura 41: Sketch usado para verificar se o botão está apertado.

Fonte: O Autor (2019)

Na Figura 41, o comando `digitalRead(botao)`; verifica o estado do botão ligado ao pino 9, armazenando a informação na variável *estado*. Se o nível lógico da variável *estado* for alto (HIGH, 1 ou *true*) a mensagem "O botão está apertado" aparecerá no monitor serial. Se o nível lógico for baixo (LOW, 0 ou *false*) a mensagem "O botão não está apertado" aparecerá no monitor serial.

4.12.1 Pull up e Pull down

O circuito da Figura 40 e o *sketch* da Figura 41 apresentam um problema. Os *push buttons* utilizam uma tensão muito baixa e, para evitar queimá-los, é necessário colocar um resistor de 10kΩ. Geralmente, quando se instala um *push button* em um circuito, o nível lógico do botão tende a ficar flutuando. Quando não apertado, o nível lógico do botão não é totalmente zero, e sim um valor próximo a zero, que oscila, o que pode ser resolvido colocando um resistor no circuito.

Se o resistor for colocado no lado do botão ligado ao pino GND, será um PULL DOWN, que significa que o nível lógico no botão será fixado em LOW. Se o resistor for colocado no lado do botão que está ligado ao pino digital do Arduino, será um PULL UP, onde o nível lógico do botão é fixado em HIGH.

Em PULL UP, quando apertado o *push button*, o nível lógico será LOW. Em PULL DOWN, quando apertado o *push button*, o nível lógico será HIGH.

Assim, o circuito da Figura 40 deve ser montado da seguinte forma:

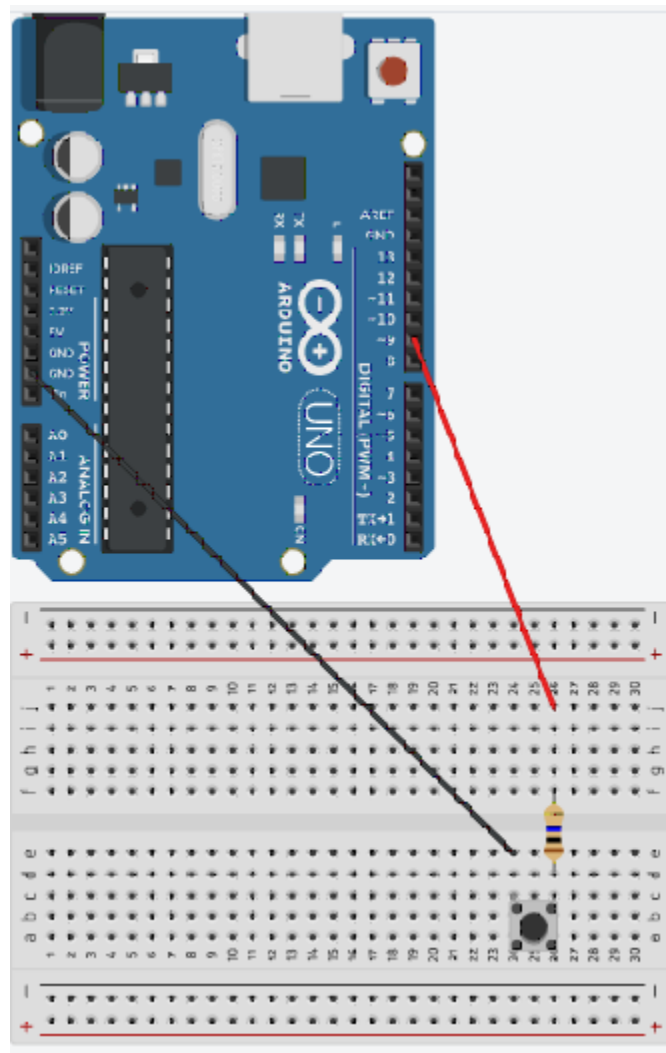


Figura 42: Circuito com botão em PULL UP.

Fonte: O Autor (2019)

Na Figura 42, o botão está em modo PULL UP, já que o resistor se encontra no lado positivo do circuito. Para colocar o botão em modo PULL DOWN, pode-se colocar o resistor entre o botão e o cabo jumper de cor preta, ligado ao pino GND do Arduino.

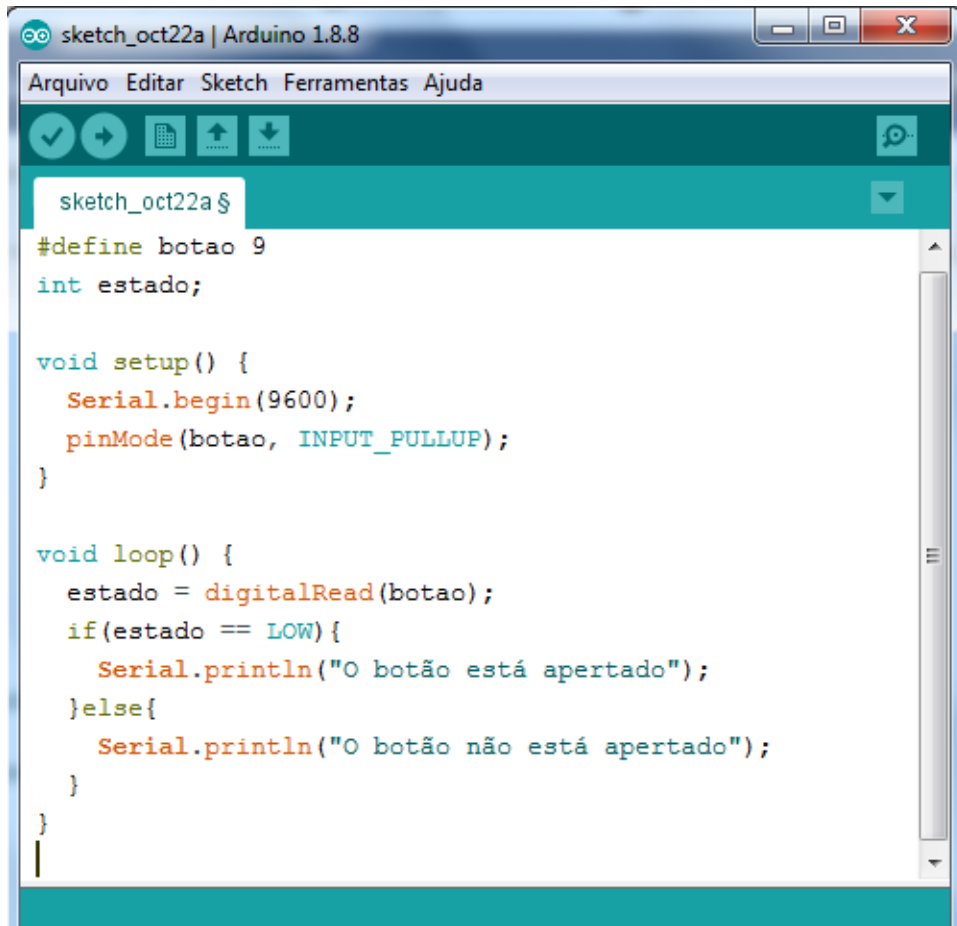
4.12.2 Comando INPUT_PULLUP

Caso o usuário não tenha resistores de 10k Ω , existe uma forma de evitar que o botão seja queimado e fixar seu nível lógico. A solução será declarar o pino ao qual o botão está ligado como INPUT_PULLUP, dentro da função *setup*.

Isso se faz de maneira igual à forma que se declara o que é INPUT e o que é OUTPUT, sendo que no caso se escreve INPUT_PULLUP no lugar de INPUT.

O comando emprega um resistor interno do Arduino, o que evita que o botão queime, estabilizando seu nível lógico em HIGH.

O código da Figura 41 pode ser reescrito da seguinte forma:

The image is a screenshot of the Arduino IDE interface. The title bar at the top reads 'sketch_oct22a | Arduino 1.8.8'. Below the title bar is a menu bar with 'Arquivo', 'Editar', 'Sketch', 'Ferramentas', and 'Ajuda'. Underneath the menu bar is a toolbar with icons for saving, running, uploading, and other functions. The main text area contains the following code:

```
sketch_oct22a $
#define botao 9
int estado;

void setup() {
  Serial.begin(9600);
  pinMode(botao, INPUT_PULLUP);
}

void loop() {
  estado = digitalRead(botao);
  if(estado == LOW){
    Serial.println("O botão está apertado");
  }else{
    Serial.println("O botão não está apertado");
  }
}
```

Figura 43: Sketch contendo botão com INPUT_PULLUP.

Fonte: O Autor (2019)

4.12.3 Comando DigitalRead()

O comando `digitalRead(botao);`, usado nos *sketchs* das Figuras 41 e 43, serve para verificar qual o estado lógico do *push button*. A palavra em inglês "*Read*" significa "Ler", e em programação significa obter informação. O termo "digital", neste caso, refere-se a um sinal que será interpretado como "ligado" ou "desligado", ou ainda "HIGH" ou "LOW". Sinais digitais são como variáveis booleanas, só admitem dois estados lógicos, não existindo meio termo.

O comando acima faz uma leitura do sinal digital no pino onde o botão está ligado. Esse comando pode retornar os valores HIGH (equivalente a 1 ou *true*) ou LOW (equivalente a 0 ou *false*).

4.13 USANDO POTENCIÔMETRO E RESISTORES

Os potenciômetros e os resistores são componentes que controlam o fluxo de corrente elétrica que passa através deles. A diferença entre eles é que, no caso do potenciômetro, a resistência pode ser controlada pelo eixo giratório, enquanto que os resistores apresentam um valor de resistência fixo (vide Tabela 2).

Assim, conforme o eixo giratório do potenciômetro é manuseado, a corrente elétrica que passa é alterada, podendo ser usado para conferir um brilho nem forte nem fraco, porém intermediário, para um LED, por exemplo.

Enquanto que os resistores possuem dois terminais e devem ser colocados no circuito onde se deseja controlar a corrente elétrica, o potenciômetro tem três terminais: um mais à esquerda, um no meio e um mais à direita. Um dos terminais nas extremidades deve ser ligado ao GND do Arduino, e o terminal do outro extremo deve ser ligado ao 5V do Arduino. No terminal do meio deve ser colocado qualquer componente eletrônico, como um LED, ou ainda pode-se ligar o terminal a um pino do próprio Arduino (Figura 44).

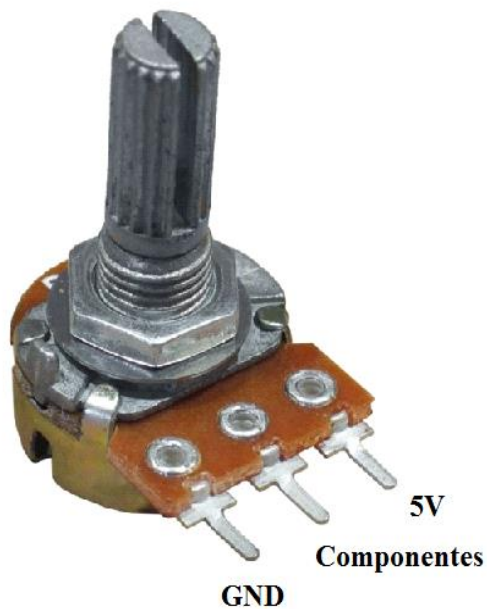


Figura 44: Potenciômetro, com indicação de onde conectar seus terminais.

Fonte: O Autor (2019)

Ao conectar um LED no terminal central do potenciômetro, é possível ver a luz do LED variar conforme o eixo giratório é manuseado. Se o terminal central do potenciômetro for ligado a um pino analógico do Arduino, será possível "ler" esse pino com o comando `analogRead`, e, a partir deste ponto, ordenar algum comando ao microprocessador.

4.14 USANDO UM LDR

O termo LDR significa *Light Dependent Resistor*, ou Resistor Dependente de Luz. O componente é um sensor de luz, cuja resistência varia em função da luminosidade do ambiente.

Diferentemente do potenciômetro, o LDR possui dois terminais ao invés de três, assemelhando-se aos resistores.

Devido à tensão que o LDR suporta, é aconselhado utilizar um resistor de 1000Ω junto do sensor, para evitar que se queime.

Ao montar o circuito com o LDR, deve-se colocar um de seus terminais ligados ao 5V do Arduino enquanto a outra extremidade estará ligada ao GND e também a um pino analógico do Arduino, conforme a Figura 45.

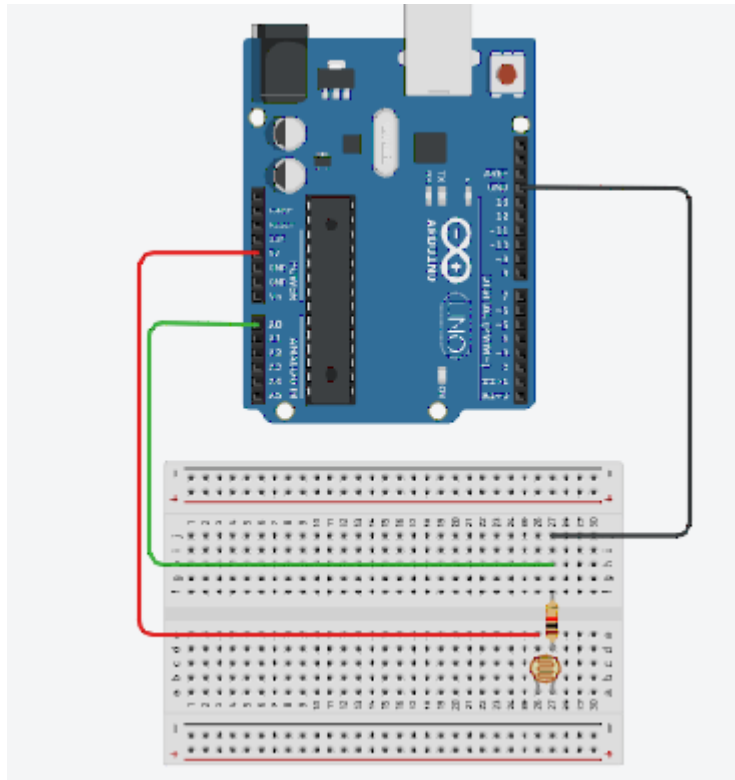


Figura 45: Circuito com LDR ligado ao pino A0.

Fonte: O Autor (2019)

4.14.1 Portas analógicas e digitais

Tanto os LDRs como muitos outros sensores geram variações em suas resistências que podem ser interpretadas pelo microprocessador para inferir algum dado. Isso significa que a partir da resistência de um LDR pode-se dizer qual a quantidade de luz em um ambiente. Para isso, é comum utilizar as portas analógicas do Arduino.

As portas analógicas são pinos indicados pela letra "A" na placa (A0, A1, A2, A3, A4, A5) onde os sinais enviados serão interpretados em uma escala de 0 a 1023. Assim, qualquer sensor que esteja ligado aos pinos terá seu sinal convertido na mesma escala. Estas portas são usadas por padrão como INPUT, sendo desnecessário declarar, na função `setup`, o comando `pinMode(, INPUT)`. No entanto, se pretende-se usar portas analógicas como OUTPUT, estas devem ser

declaradas como tal usando o comando `pinMode(, OUTPUT)`, e atuarão como portas digitais. O comando usado para se obter alguma informação das portas analógicas é o `analogRead`.

As portas digitais são os pinos indicados por números somente (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13) onde os sinais recebidos são interpretados de forma diferente em relação às portas analógicas. Neste caso, não há escala alguma, e os sinais serão 0 ou 1, o que indicaria "desligado" ou "ligado", ou ainda 0V (zero volts) ou 5V (cinco volts). O comando usado para se obter alguma informação das portas é o `digitalRead`. Para atribuir algum valor aos pinos, se usa o comando `digitalWrite`.

4.14.2 Portas PWM

Enquanto que as portas analógicas convertem a voltagem (de 0 a 5V) em uma escala de 0 a 1023, algumas portas digitais podem se comportar de forma semelhante. É o caso das portas PWM, que significa *Pulse Width Modeling*, indicadas pelo sinal “~”, na placa Arduino. No Arduino UNO, as portas PWM são as de número 3, 5, 6, 9, 10, e 11.

Os pinos PWM aceitam comandos como o `analogwrite`, visto que o pulso das portas varia, gerando uma espécie de sinal analógico. Cada vez que a porta desliga (0V) e liga (5V) se chama este movimento de pulso (Figura 46). As portas PWM ficam desligando e ligando muito rapidamente, e controlam o período em que se encontram em 5V ou em 0V. A partir desse mecanismo o Arduino pode interpretar tensões intermediárias, baseado na largura dos pulsos. Assim, a voltagem em pinos PWM é convertida em uma escala de 0 a 255, de forma análoga à escala de 0 a 1023 das portas analógicas. O valor mínimo das escalas representa 0V e o valor máximo representa 5V.

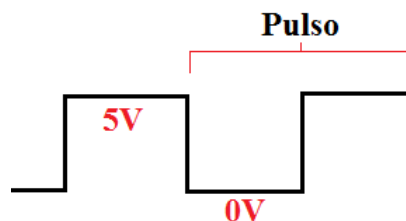


Figura 46: Pulso.

Fonte: O Autor (2019)

4.14.3 Comando map

Se for usado o comando `analogRead` em um pino analógico e se objetiva converter o valor obtido em Volts, pode-se usar o comando `map`. Observe a Figura 46, a seguir.

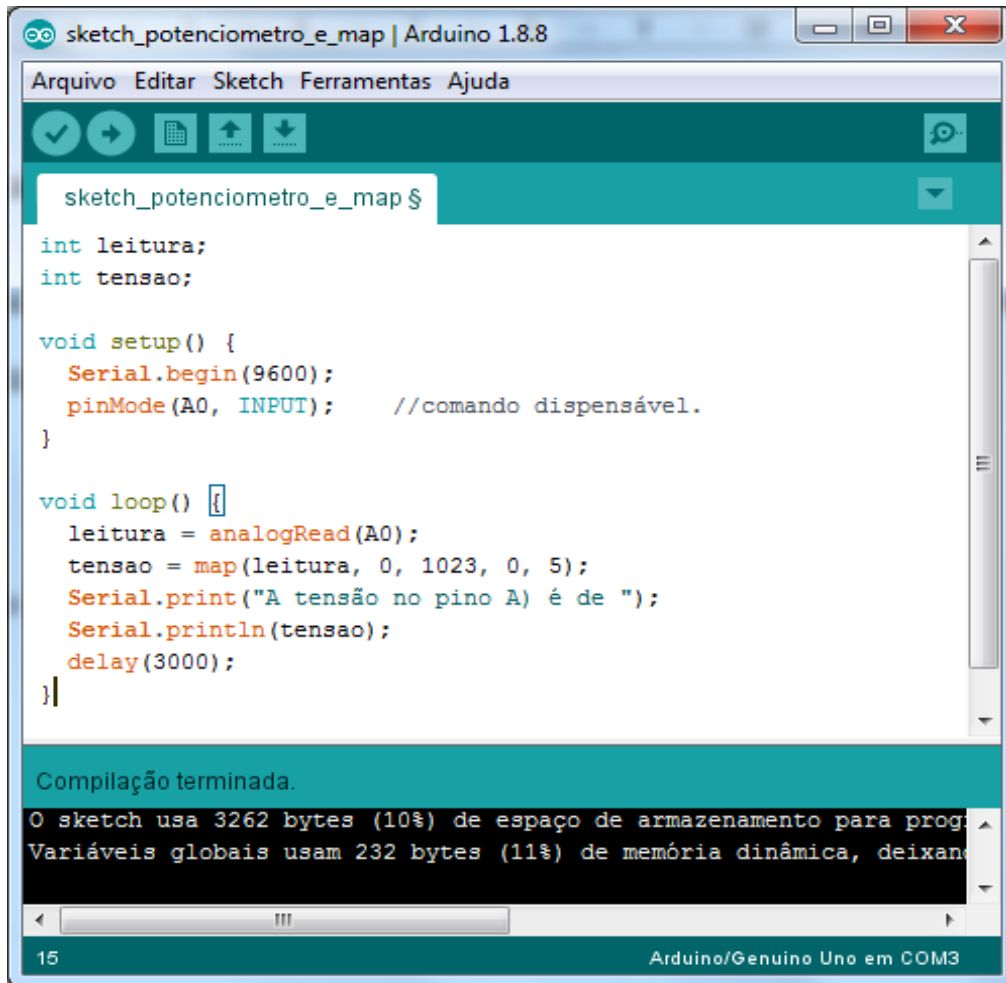


Figura 47: sketch usando o comando map.

Fonte: o Autor (2019)

O comando *map* serve para transferir um valor, em uma escala, para outra escala. São requeridas as seguintes informações: o valor que se quer transferir, o valor mínimo da escala em que ele se encontra, o valor máximo da escala em que ele se encontra, o valor mínimo da nova escala, o valor máximo da nova escala. Os dados devem ser colocados exatamente nesta ordem.

Assim, de acordo com o comando `tensao = map(leitura, 0, 1023, 0, 5)`, na Figura 46, a variável “leitura” é transferida de uma escala de 0 a 1023 para uma escala de 0 a 5, que representa a tensão, em Volts. O resultado é guardado na variável “tensao”.

Como pode ser observado, as variáveis usadas são do tipo inteiras, ou seja, não admitem números decimais. A razão é que o comando *map* só trabalha com números inteiros, podendo gerar algum erro se forem usadas variáveis do tipo *float*, por exemplo.

4.15 CAPACITORES

4.15.1 Por que usar capacitores?

Os capacitores são componentes que, uma vez colocados no circuito elétrico, estabilizam o fluxo de energia, evitando oscilações. Isto ocorre uma vez que os capacitores podem armazenar carga elétrica, liberando-a gradualmente para o restante do circuito, de forma controlada. Assim, os capacitores podem ser posicionados antes de outros componentes eletrônicos, fazendo com que a energia que chega aos componentes seja constante, favorecendo um melhor funcionamento do circuito.

Alguns capacitores possuem um terminal positivo e outro negativo. Quando ocorre, o terminal positivo deve ser colocado na parte positiva do circuito elétrico, enquanto que o terminal negativo deve ser colocado na parte ligada ao GND do circuito elétrico.

Muitos componentes funcionam melhor quando utilizados em conjunto com capacitores, como por exemplo *push buttons*.

4.15.2 Push button e o efeito bouncing

Uma característica dos *push buttons* é que, quando são apertados, não mudam instantaneamente do estado LOW para HIGH, e quando se deixa de apertá-los não mudam instantaneamente do estado HIGH para LOW. O que ocorre é que, sempre em momentos de transição, o sinal oscila, ou seja, quando se aperta o *push button*, o sinal oscila algumas vezes entre HIGH e LOW até se estabilizar em HIGH, e quando se deixa de apertar o *push button* este

fenômeno ocorre mais uma vez, até se estabilizar em LOW. A oscilação se dá em velocidade muito alta, e o fenômeno é chamado de *bouncing*.

Quando se quer acender um LED ao apertar um *push button*, o efeito *bouncing* não será tão prejudicial, já que a oscilação é muito rápida e não é possível perceber a variação no LED, a olho nu. Entretanto, em alguns casos, o efeito *bouncing* deve ser controlado, evitando problemas no circuito.

Por exemplo, para acender um LED após apertar um *push button* dez vezes, e desligá-lo após apertar mais dez vezes o mesmo *push button*, ocorrerão problemas. A cada vez que se aperta o botão, o microprocessador interpreta como se o botão tivesse sido apertado mais de uma vez, devido ao efeito *bouncing*, fazendo com que o LED acenda e apague antes do momento desejado.

Para resolver o inconveniente, existem duas maneiras:

1. Usar um capacitor junto ao *push button*, estabilizando o fluxo de energia que chega, evitando a oscilação entre HIGH e LOW;
2. Usando a programação para evitar o efeito *bouncing*.

Usando um capacitor, o circuito ficaria montado conforme a Figura 48.

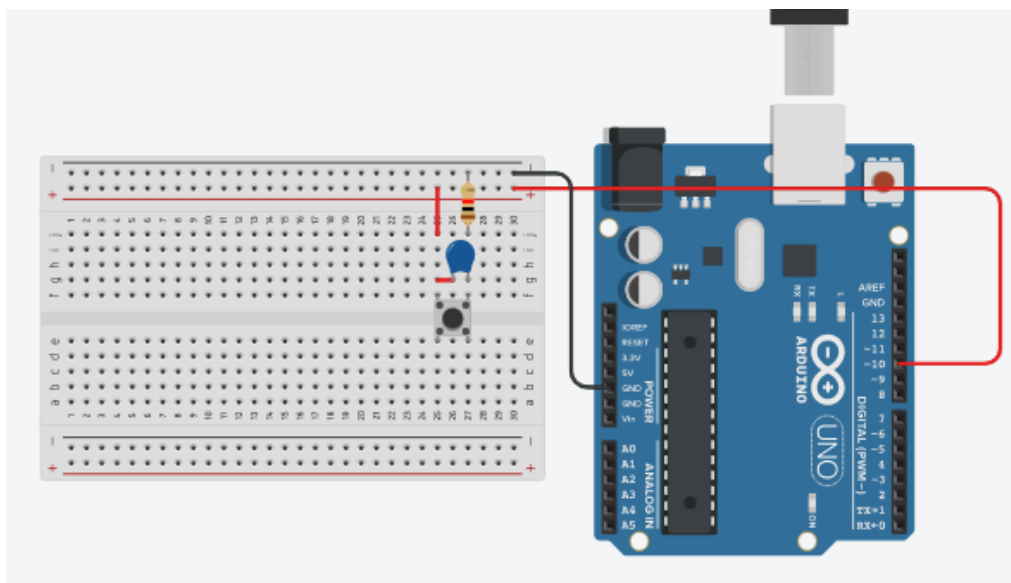
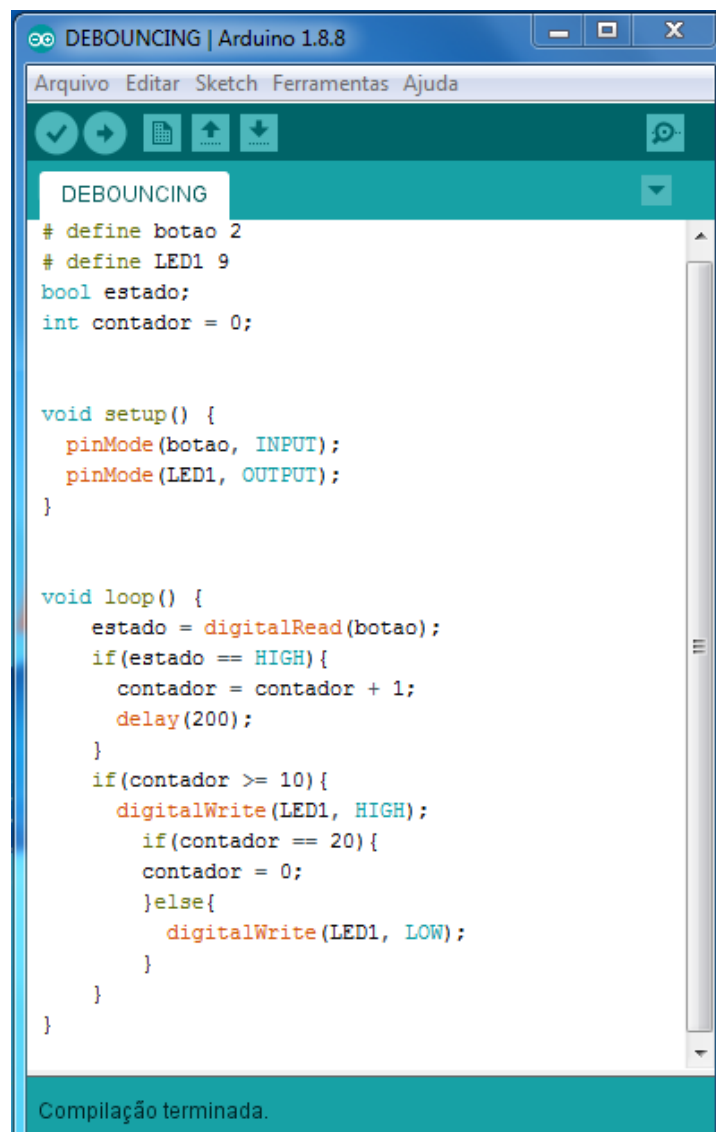


Figura 48: Circuito com capacitor junto a *push button*.

Fonte: O Autor (2019)

Na Figura 48, o capacitor estabiliza a corrente, que chega então no botão de forma constante, evitando o efeito *bouncing*.

Para resolver o efeito *bouncing* com a própria programação, existem inúmeros caminhos. O que se faz, geralmente, é ordenar ao microprocessador que espere alguns milésimos de segundo sempre que o botão mudar de estado HIGH para LOW, ou vice versa. Essa medida faz com que as oscilações sejam ignoradas. O ato de corrigir o efeito *bouncing* é chamado de *debouncing*. Observe a Figura 49.

The image is a screenshot of the Arduino IDE interface. The title bar reads 'DEBOUNCING | Arduino 1.8.8'. The menu bar includes 'Arquivo', 'Editar', 'Sketch', 'Ferramentas', and 'Ajuda'. Below the menu bar is a toolbar with icons for saving, running, uploading, and other functions. The main text area contains the following C++ code:

```
# define botao 2
# define LED1 9
bool estado;
int contador = 0;

void setup() {
  pinMode(botao, INPUT);
  pinMode(LED1, OUTPUT);
}

void loop() {
  estado = digitalRead(botao);
  if(estado == HIGH){
    contador = contador + 1;
    delay(200);
  }
  if(contador >= 10){
    digitalWrite(LED1, HIGH);
    if(contador == 20){
      contador = 0;
    }else{
      digitalWrite(LED1, LOW);
    }
  }
}
```

At the bottom of the window, a status bar indicates 'Compilação terminada.' (Compilation finished.).

Figura 49: Primeira tentativa de debounce.

Fonte: O Autor (2019)

Na Figura 49 há uma variável chamada “contador”, que vale inicialmente 0 e é acrescido de 1 sempre que o *push button* for apertado. Quando o valor da variável alcança 10, o LED acende (variável LED1 em estado HIGH), indicando que o botão foi apertado dez vezes. Se a variável “contador” alcança o valor 20, isso indica que o botão foi apertado vinte vezes, e o LED irá se apagar (variável LED1 em estado LOW).

Para que a contagem das vezes que o *push button* foi apertado funcione, se faz necessário o comando `delay(200)`, que obriga o microprocessador a esperar 200 milissegundos sempre que o estado do botão mudar de LOW para HIGH. A princípio parece funcionar, porém se o botão for pressionado por um período longo de tempo, o microprocessador interpretará como se o botão estivesse sendo apertado várias vezes, e não uma somente.

Assim, a Figura 50 traz uma outra possibilidade de *debouncing*. Neste *sketch* temos alguns comandos novos, que serão explicados adiante.

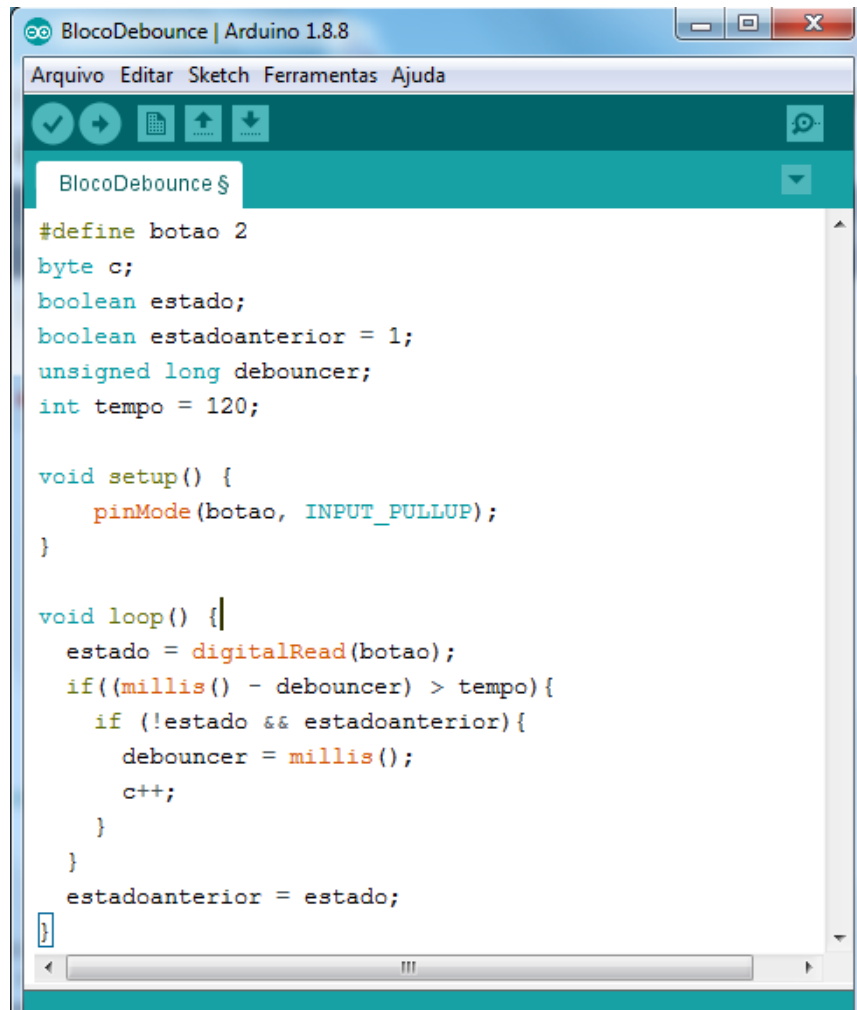


Figura 50: Segunda tentativa de debouncing.

Fonte: O Autor (2019)

Primeiramente, é necessário esclarecer para que serve a função `millis()`. O Arduino tem a capacidade de contar o tempo que transcorre desde o momento em que é ligado, na unidade de tempo milissegundos, até certo valor, reiniciando a contagem. A função `millis()` mostra justamente a contagem. Em outras palavras, `millis()` representa um determinado momento, em milissegundos.

Na Figura 50 há duas condicionais em sequência. A variável “estado” verifica se o *push button* está apertado ou não (LOW ou HIGH). Se o momento atual subtraído do momento em que o estado do botão mudou for maior que a variável “tempo” (que vale 120 milissegundos), será executado a outra condicional, que verifica se o estado atual do botão é LOW e se seu estado

anterior era HIGH. Em caso positivo, a variável “c” será acrescida de 1, funcionando como um contador de quantas vezes o botão já foi apertado.

O código é um dos muitos códigos que servem para fazer o *debouncing*. Outros códigos, tão eficientes quanto o apresentado na Figura 50 podem ser encontrados na internet, notadamente no próprio *site* do Arduino. Ao executar *debounce*, sempre que o botão é apertado, independentemente do tempo que se demora, o Arduino entenderá que o botão foi apertado uma só vez, possibilitando contar quantas vezes o botão é apertado sem erros.

Como foi dito no início do presente tópico, o efeito *bounce* pode ser resolvido também com a ajuda de capacitores, que armazenam a carga elétrica e estabilizam o fluxo de energia que chega aos componentes eletrônicos. Cabe ao usuário decidir como resolverá o efeito *bounce*, de acordo com os recursos que tem e com seus conhecimentos em programação.

4.16 FUNÇÕES

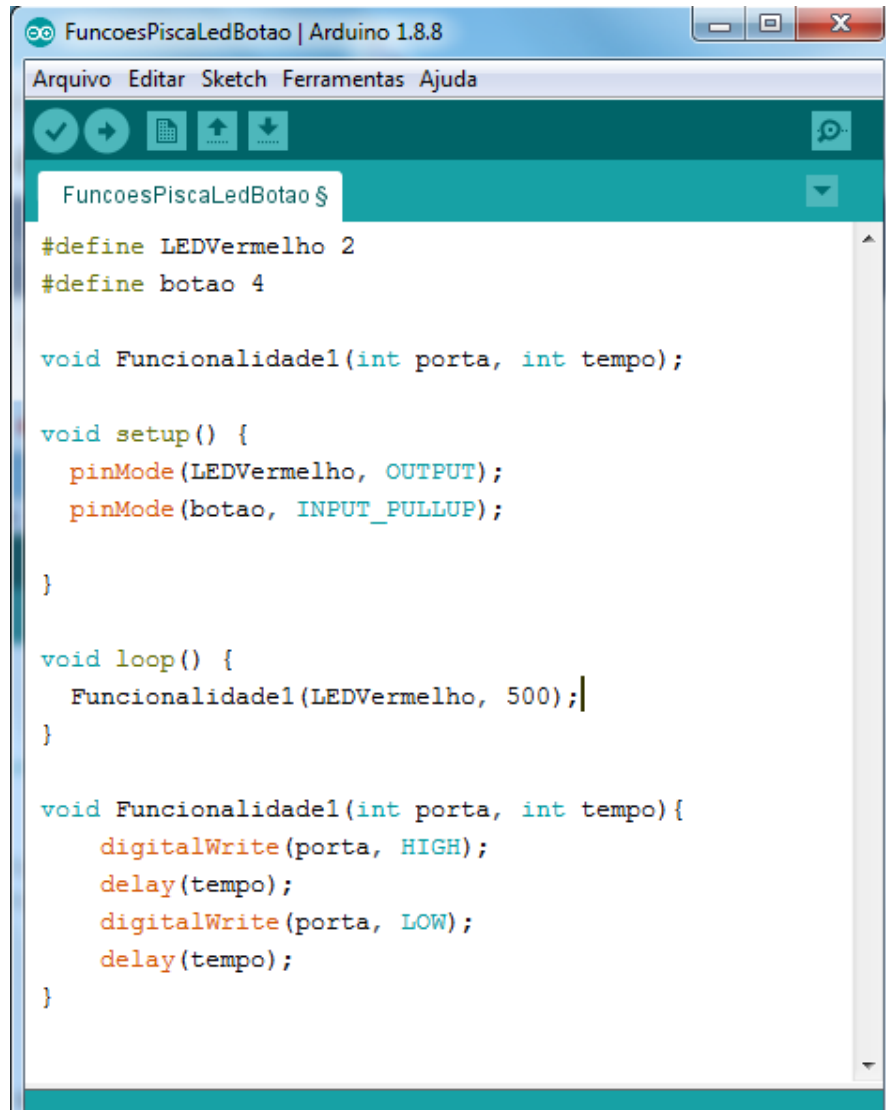
Funções são também chamadas de sub-rotinas, e representam um conjunto de comandos que executam alguma tarefa. Pode ser definido, ainda, como um atalho para um bloco de comandos.

As funções são seguidas de parênteses, dentro dos quais devem ser colocados os parâmetros necessários para que sejam executadas. Até este ponto do curso, já foram vistas algumas funções. Como exemplo, temos o `void loop()`, `void setup()`, `map()`, `print()`, `println()`, `Serial.begin()`, `digitalRead()`, `analogWrite()`, entre muitas outras. Todos os comandos seguidos por parênteses são funções. Dentro dos parênteses, é informado algum dado necessário para o funcionamento da função. Por exemplo, na função `analogWrite()` deve-se informar o número do pino em que a função se aplicará, enquanto que em `Serial.begin()` deve-se informar a velocidade da comunicação com o monitor serial.

Eventualmente, se faz necessário usar funções que o Arduino não reconhecerá, exclusivas de algumas bibliotecas. Neste caso, deve-se incluir as bibliotecas no início do *sketch*.

As funções podem também ser criadas pelo usuário. Para criá-las, primeiro é necessário declará-las antes do `void setup()`. Nesta etapa, escreve-se o tipo da função, seguida de seu nome e seus parâmetros, dentro de parênteses. Depois, ao final do *sketch*, deve-se definir as funções

declaradas, repetindo o comando acima e abrindo chaves. Dentro das chaves, deve-se especificar tudo o que a função deverá fazer, e qual será seu retorno, se tiver algum. Funções sem retorno serão sempre do tipo `void`. Observe a Figura 51, a seguir.

The image is a screenshot of the Arduino IDE interface. The title bar reads 'FuncoesPiscaLedBotao | Arduino 1.8.8'. The menu bar includes 'Arquivo', 'Editar', 'Sketch', 'Ferramentas', and 'Ajuda'. Below the menu bar is a toolbar with icons for saving, running, uploading, and downloading. The main text area contains the following code:

```
FuncoesPiscaLedBotao $
#define LEDVermelho 2
#define botao 4

void Funcionalidade1(int porta, int tempo);

void setup() {
  pinMode(LEDVermelho, OUTPUT);
  pinMode(botao, INPUT_PULLUP);
}

void loop() {
  Funcionalidade1(LEDVermelho, 500);
}

void Funcionalidade1(int porta, int tempo){
  digitalWrite(porta, HIGH);
  delay(tempo);
  digitalWrite(porta, LOW);
  delay(tempo);
}
```

Figura 51: Declaração e definição da função chamada “Funcionalidade1”.

Fonte: O Autor (2019)

Na Figura 51, a função declarada chama-se “Funcionalidade1”, e pode ser usada em qualquer momento no *sketch*. Observe a Figura 52:



```
aula_12_exercicio_de_funcoes $
# define botao 2
# define LED1 9
# define LED2 10
# define LED3 11
byte c;
boolean estado;

void setup() {
  Serial.begin(9600);
  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);
  pinMode(LED3, OUTPUT);
  pinMode(botao, INPUT_PULLUP);
}

void loop() {
  estado = digitalRead(botao);
  if (!estado){
    c++;
    Serial.println(c);
  }
  if (c >= 5){
    Funcionalidade1();
  }
}
Compilação terminada.
```

Figura 52: Sketch onde se utiliza uma função criada pelo próprio usuário.

Fonte: O Autor (2019)

Na Figura 52 utiliza-se a função declarada na Figura 51, que fará com que os três LEDs (LED1, LED2 e LED3) acendam em sequência, em intervalos de 300 milissegundos, apagando todos os LEDs ao final e recomeçando o ciclo.

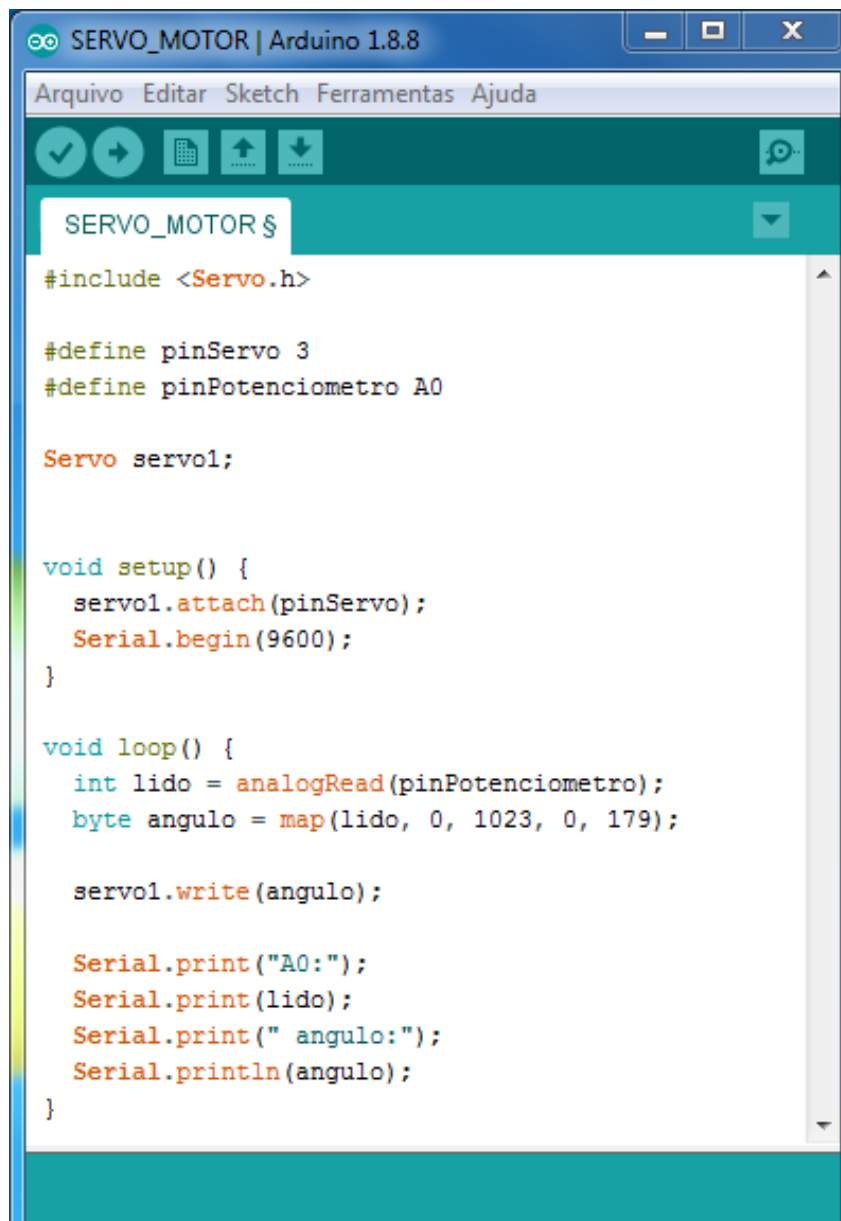
4.17 SERVO MOTOR E BIBLIOTECAS

O servo motor é um pequeno motor, que pode girar 180° ou 360°. Os componentes podem ser usados em diversos projetos, como por exemplo um carrinho de controle remoto. Neste caso, os servos motores podem fazer as rodas girarem, permitindo o movimento do carro.

Geralmente os servos motores possuem três terminais, sendo que um é ligado ao GND do Arduino, outro é ligado ao 5V do Arduino, e o terceiro terminal é ligado a uma porta PWM do Arduino. Para controlar o componente com facilidade, no entanto, se faz necessário usar uma biblioteca.

Bibliotecas são como grupos de códigos que podem ser compartilhados, possuindo diversos comandos úteis. O foco desta apostila não é a criação de bibliotecas, portanto não será detalhado como criá-las. Basta saber que, no Arduino, quando se cria uma biblioteca se utiliza o comando `Class`, onde se cria uma classe. As classes representam grupos de componentes, como, por exemplo, uma classe de LEDs, de botões ou de um módulo *Bluetooth*. Dentro das classes haverá propriedades e métodos, que serão como as variáveis e as funções, respectivamente. Para compreender melhor o tema é recomendado estudar programação orientada a objetos.

Para usar uma biblioteca, no início do *sketch* deve-se inserir o comando `#include <nome_da_biblioteca.extensão>`. Quando se quer informar que algum componente é um objeto pertencente a essa biblioteca, escreve-se o nome da biblioteca seguido do nome do objeto. Feito o procedimento, pode-se usar os comandos da biblioteca. Observe a Figura 53.

The image shows a screenshot of the Arduino IDE interface. The title bar at the top reads 'SERVO_MOTOR | Arduino 1.8.8'. Below the title bar is a menu bar with 'Arquivo', 'Editar', 'Sketch', 'Ferramentas', and 'Ajuda'. Underneath the menu bar is a toolbar with icons for checking, running, saving, and other functions. The main text area displays the following C++ code:

```
SERVO_MOTOR $  
  
#include <Servo.h>  
  
#define pinServo 3  
#define pinPotenciometro A0  
  
Servo servol;  
  
void setup() {  
    servol.attach(pinServo);  
    Serial.begin(9600);  
}  
  
void loop() {  
    int lido = analogRead(pinPotenciometro);  
    byte angulo = map(lido, 0, 1023, 0, 179);  
  
    servol.write(angulo);  
  
    Serial.print("A0:");  
    Serial.print(lido);  
    Serial.print(" angulo:");  
    Serial.println(angulo);  
}
```

Figura 53: Sketch para controlar servo motor com um potenciômetro.

Fonte: O Autor (2019)

Na Figura 53 acima, a biblioteca “Servo” possui extensão “.h” e é incluída com o comando “#include”. O objeto “servol” é declarado como pertencente à biblioteca “Servo”. Na biblioteca, deve-se declarar onde o objeto “servol” está conectado (nesse caso, no pino 3), usando o comando `servol.attach`. O comando `servol.write()` serve para indicar qual o ângulo que o motor irá rodar. Ao converter o valor obtido com o comando `analogRead()`, de uma escala de 0 a 1023 para

uma escala de 0 a 179, temos os ângulos que serão usados pelo servo motor. Assim, sempre que o potenciômetro for girado, o servo motor rodará também.

Observe que sempre que se utiliza algum método da biblioteca, antes deverá vir o nome do objeto. Observe a Figura 54:

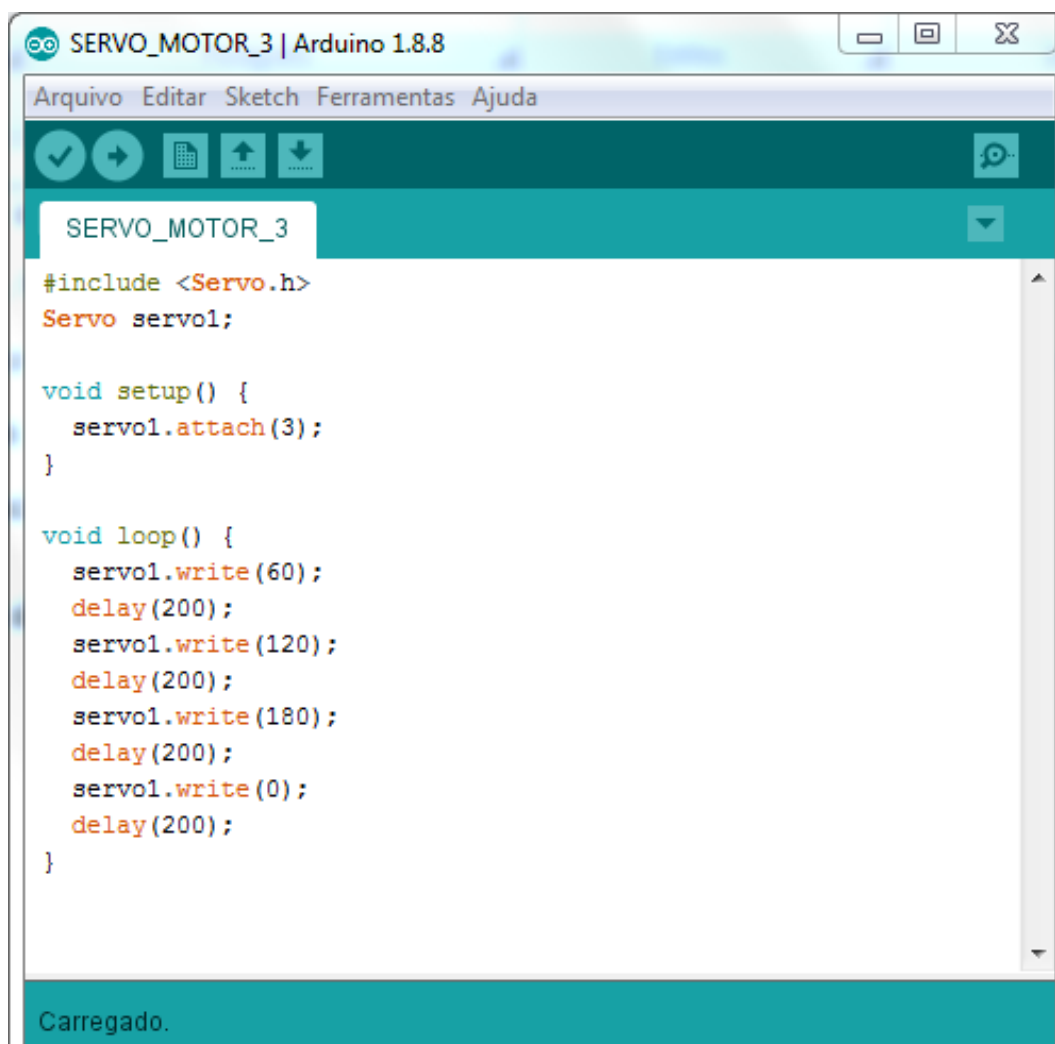


Figura 54: Programa que roda o motor nos ângulos 60, 120, 180 e 0°.

Fonte: O Autor (2019)

Como pode ser visto na Figura 54 e também na Figura 53, o passo a passo para se utilizar bibliotecas é sempre: (I) incluí-las no início do *sketch* (`#include <Servo.h>`), (II) declarar os objetos que utilizarão a biblioteca (`Servo servo1;`), e (III) usar os métodos desejados sempre com o nome do objeto os precedendo (`servo1.attach()` e `servo1.write()`).

4.18 DISPLAY LCD

Existem inúmeros Displays para Arduino, que servem para mostrar mensagens para o usuário, possuindo grande importância em projetos mais avançados, principalmente aqueles que envolvem sensores e monitoramento de alguma variável (luz, temperatura, umidade, entre outros).

Nesta apostila, lida-se com o Display LCD de 16 pinos. Para usá-lo, é necessário incluir a biblioteca `LiquidCrystal.h`. Observe a relação dos terminais do Display e onde devem ser conectados abaixo:

- 1° VSS-----GND do Arduino
- 2° VDD-----5V do Arduino
- 3° VO-----Potenciômetro ou GND do Arduino
- 4° RS-----Pino de controle ligado à Porta Digital
- 5° RW-----Pino de controle ligado à Porta Digital
- 6° E-----Pino de controle, ligado à Porta Digital
- 7° D0 até D7-----Pinos de dados, ligados à Portas Digitais
- 8° A-----Ânodo, ligado ao positivo, com resistor de 220Ω
- 9° K-----Cátodo, ligado ao negativo (GND)

O pino VO tem a função de controlar a luz das letras, influenciando no contraste das mesmas com o fundo do Display. Se o pino VO for ligado ao GND, a cor das letras será a mais clara possível. Através do pino RS pode-se informar ao Arduino se o Display deve apagar, ligar, piscar, entre outros comandos. O pino RW é por onde se informa se está sendo lido ou recebendo informações no Display. O pino E significa “enable”, e serve para o Arduino informar que está enviando informações para o Display. Os pinos de dados (D0, D1, D2, D3, D4, D5, D6, D7) são os responsáveis por enviar as mensagens que devem aparecer no Display. Os dois últimos pinos (A e K) controlam a luz de fundo, que consiste em um LED posicionado atrás do Display. Por se tratar de um LED, é necessário colocar um resistor de 220Ω para que não queime.

A programação de um circuito com um Display pode ser observada na Figura 55, a seguir.



Figura 55: Programa usando Display LCD.

Fonte: O Autor (2019)

Na Figura 55, a biblioteca `LiquidCrystal.h` é incluída usando o comando `#include`. Logo em seguida, o objeto `LCD` é declarado como pertencente à biblioteca, através do comando `LiquidCrystal LCD(2,3,4,6,7,8,9,10,11,12,13);`. Note que, após o nome do objeto (`LCD`), foram colocados números entre os parênteses. Os números são parâmetros, que representam os pinos em que os terminais do Display foram conectados. Devem ser informados os pinos onde os terminais RS, RW, E e os pinos de dados estão ligados. O comando `LCD.begin(16,2)` inicia a comunicação com o Display, sendo necessário para utilizá-lo. Os números dentro dos parênteses representam a quantidade de colunas e de linhas do Display, respectivamente. O comando `LCD.print()` serve para informar o Display qual mensagem deverá ser exibida. As mensagens sempre deverão vir entre aspas. Por fim, o comando `LCD.clear()` serve para limpar a mensagem exibida anteriormente.

Note que os comandos `clear` e `print` só podem ser usados porque a biblioteca `LiquidCrystal.h` foi incluída e, assim como quando se utiliza o servo motor, o Display LCD precisa de bibliotecas específicas para facilitar sua programação. Muitos componentes precisarão de bibliotecas para que a programação fique facilitada, e para utilizar os recursos de forma eficiente deve-se ter em mente os seguintes passos:

- 1º Incluir biblioteca com o comando `#include`;
- 2º Declarar um objeto que pertencerá à biblioteca (comando `LiquidCrystal LCD`, nesse caso). Relembrando: o nome da biblioteca precede o nome do objeto, seguido de eventuais parâmetros entre parênteses;
- 3º Usar os métodos da biblioteca, com o nome do objeto os precedendo. No caso dos comandos `LCD.print` e `LCD.clear`, o termo “LCD” é o nome do objeto, e os termos “print” e “clear” são os métodos da biblioteca utilizada.

4.19 CONSIDERAÇÕES FINAIS SOBRE PROGRAMAÇÃO PARA ARDUINO

No capítulo IV desta apostila foi possível abordar a programação do Arduino e como usar alguns de seus componentes. Os comandos apresentados até este ponto permitem que projetos simples sejam criados. As funções, apresentadas no tópico 4.16, podem otimizar o programa, tornando-o mais simples e organizado. As bibliotecas, apresentadas no tópico 4.17, facilitam a utilização de muitos componentes eletrônicos, também simplificando o trabalho do programador. Programadores experientes podem, inclusive, criar suas próprias bibliotecas. No entanto, é necessário deter um conhecimento mais aprofundado em programação, com destaque para a programação orientada a objeto, assunto que não foi abordado nesta apostila básica.

No entanto, muito já pode ser feito pelos alunos deste curso e, para que haja um verdadeiro aprendizado – concreto e duradouro – faz-se necessário praticar.

Qualquer biblioteca pode ser utilizada, basta compreender seus comandos. Desta forma, ao se procurar saber o que já foi feito com Arduino, se descobrirá que a capacidade desta tecnologia é muito ampla, permitindo projetos de automação residencial, de robótica, dentre inúmeros outros. O alcance do Arduino é gigantesco, e a imaginação do programador é o principal fator limitante do que se poder fazer com a eletrônica.

Assim, seja criativo e imaginativo, nunca parando de estudar, porque muito pode ser feito.

[illegible]

This image shows a blank sheet of white paper with horizontal ruling lines. The lines are evenly spaced and run across the width of the page. There are no margins, text, or other markings on the paper.

[illegible]

This image shows a single sheet of white paper with horizontal ruling lines. The lines are evenly spaced and run across the width of the page. There are no margins, text, or other markings on the paper.

This image shows a single sheet of white paper with horizontal ruling lines. The lines are evenly spaced and run across the width of the page. There are no margins, text, or other markings on the paper.