



Trabalho de Conclusão de Curso

**ArduEasy - Uma biblioteca para abstração de
hardware e controle de tempo na plataforma
Arduino**

Jhonatan Mickael Silva Nascimento

Orientador:

Lucas Benevides Viana de Amorim

Maceió, Maio de 2025

Jhonatan Mickael Silva Nascimento

ArduEasy - Uma biblioteca para abstração de hardware e controle de tempo na plataforma Arduino

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação do Instituto de Computação da Universidade Federal de Alagoas.

Orientador:

Lucas Benevides Viana de Amorim

Maceió, Maio de 2025

Catálogo na fonte
Universidade Federal de Alagoas
Biblioteca Central
Divisão de Tratamento Técnico
Bibliotecária: Girlaine da Silva Santos – CRB-4 – 1127

N244a Nascimento, Jhonatan Mickael Silva.

ArduEasy: uma biblioteca para abstração de hardware e controle de tempo na plataforma Arduino / Jhonatan Mickael Silva Nascimento. – 2025.
72 f.: il.

Orientador: Lucas Benevides Viana de Amorim.

Monografia (Trabalho de Conclusão em Ciência da Computação:
Bacharelado) - Universidade Federal de Alagoas, Instituto de Computação.
Maceió, 2025.

Bibliografia: f. 58- 60.

Apêndice: f. 61- 72.

1. Arduino (Controlador programável). 2. Abstração de hardware. 3.
programação orientada a objetos (Computação). 4. Prototipagem eletrônica. I.
Título.

CDU: 004.41

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação do Instituto de Computação da Universidade Federal de Alagoas, aprovada pela comissão examinadora que abaixo assina.



Documento assinado digitalmente
LUCAS BENEVIDES VIANA DE AMORIM
Data: 10/06/2025 13:48:36-0300
Verifique em <https://validar.iti.gov.br>

Lucas Benevides Viana de Amorim - Orientador
Universidade Federal de Alagoas



Documento assinado digitalmente
ERICK DE ANDRADE BARBOZA
Data: 10/06/2025 10:34:56-0300
Verifique em <https://validar.iti.gov.br>

Erick de Andrade Barboza - Examinador
Instituto de Computação
Universidade Federal de Alagoas



Documento assinado digitalmente
GLAUBER RODRIGUES LEITE
Data: 10/06/2025 19:23:19-0300
Verifique em <https://validar.iti.gov.br>

Glauber Rodrigues Leite - Examinador
Instituto de Computação
Universidade Federal de Alagoas

Agradecimentos

Agradeço primeiramente a Deus, por me conceder força, sabedoria e perseverança para completar esta jornada. À minha mãe, Vanderléa, pelo amor incondicional, apoio incansável e incentivo em todos os momentos. À minha tia Lenna, e às minhas irmãs, Aysla e Amyne, pelo carinho, confiança e suporte que sempre me fortaleceram. Aos meus avós, Antônio e Calina, que partiram, mas deixaram um legado eterno em meu coração, inspirando-me com seus exemplos de vida e carinho. Aos demais familiares, que com suas palavras de encorajamento foram essenciais para minha trajetória.

Aos meus amigos e colegas de curso, agradeço pela amizade, pelas trocas de conhecimento, pelos momentos de descontração e pelo aprendizado coletivo, que tornaram esta experiência acadêmica tão especial.

Um agradecimento especial ao meu orientador, Lucas, pela dedicação, paciência e orientação fundamental, cuja expertise e comprometimento foram indispensáveis para a realização deste trabalho. Aos professores do Instituto de Computação, pela transmissão de conhecimento e apoio ao longo de minha formação, contribuindo significativamente para meu crescimento acadêmico e pessoal. À Universidade Federal de Alagoas, por proporcionar a estrutura e o ambiente que tornaram este trabalho possível.

Aos participantes do questionário, minha gratidão por sua colaboração e contribuições valiosas, que enriqueceram os resultados desta pesquisa.

A todos, minha mais profunda gratidão, pois cada um, à sua maneira, foi peça-chave para que este trabalho se concretizasse.

Resumo

Este trabalho apresenta o desenvolvimento da biblioteca ArduEasy, uma ferramenta de código aberto projetada para facilitar a programação na plataforma Arduino, especialmente para iniciantes. A biblioteca tem como objetivo abstrair a manipulação de hardwares básicos, como LEDs, botões, motores DC e teclados matriciais, além de oferecer suporte ao controle de tempo e execução não bloqueante. Implementada utilizando o paradigma de programação orientada a objetos, a ArduEasy permite que os usuários foquem na lógica do projeto, reduzindo a complexidade associada à manipulação dos pinos do Arduino e ao gerenciamento de temporização. O estudo incluiu experimentos comparativos entre a ArduEasy e a API padrão do Arduino, com questionários aplicados a usuários para avaliar sua usabilidade e eficiência. Os resultados indicam que a ArduEasy reduz significativamente a curva de aprendizado e melhora a experiência de desenvolvimento, especialmente para iniciantes, ao proporcionar uma interface mais intuitiva e funcionalidades que evitam bloqueios no código. A biblioteca foi disponibilizada em um repositório público, contribuindo para a comunidade Arduino e promovendo a acessibilidade na prototipagem eletrônica.

Palavras-chave: Arduino, biblioteca, abstração de hardware, controle de tempo, programação orientada a objetos, prototipagem eletrônica.

Abstract

This work presents the development of ArduEasy, an open-source library designed to simplify programming on the Arduino platform, particularly for beginners. The library aims to abstract the manipulation of basic hardware components, such as LEDs, buttons, DC motors, and matrix keypads, while providing support for time control and non-blocking execution. Implemented using the object-oriented programming paradigm, ArduEasy enables users to focus on project logic, reducing the complexity associated with Arduino pin manipulation and timing management. The study included comparative experiments between ArduEasy and the standard Arduino API, with questionnaires administered to users to assess usability and efficiency. The results indicate that ArduEasy significantly reduces the learning curve and enhances the development experience, especially for novices, by offering a more intuitive interface and features that prevent code blocking. The library has been made available in a public repository, contributing to the Arduino community and promoting accessibility in electronic prototyping.

Key-words: Arduino, library, hardware abstraction, time control, object-oriented programming, electronic prototyping.

Lista de Figuras

2.1	Arquitetura Harvard	6
2.2	Arquitetura von Neumann	6
2.3	Sensor de luminosidade LDR	8
2.4	Sensor de temperatura e humidade DHT11	8
2.5	Led	9
2.6	Motor de corrente contínua	9
2.7	Motor Shield	9
2.8	Módulo controlador de Motor	10
2.9	Placa Arduino UNO	11
2.10	Pinagem do ATmega328P	12
3.1	Diagrama de camadas	16
3.2	Diagrama de pacotes	17
3.3	Diagrama de classes dos componentes passivos	18
3.4	Diagrama de classes dos componentes ativos	21
3.5	Ligação interna de um teclado matricial 4x4	32
3.6	Pinagem de um teclado matricial 4x4	33
4.1	Modulo semáforo	43
4.2	Modulo botão	43
4.3	Protótipo de um semáforo	44
5.1	Gráficos das respostas do formulário de avaliação da ArduEasy	49
5.2	Gráficos das respostas do formulário de avaliação da API padrão	53
5.3	Gráficos das respostas do formulário de avaliação comparativa	55

Lista de Quadros

5.1	Respostas de “Qual foi o maior benefício de usar a biblioteca ArduEasy?” . . .	50
5.2	Respostas de “Alguma parte da tarefa foi particularmente difícil? Por quê?” . .	50
5.3	Respostas de “Quais funcionalidades da ArduEasy você considerou mais úteis?”	51
5.4	Respostas de “Qual foi o maior desafio que você enfrentou ao usar a API padrão do Arduino?”	54
5.5	Respostas de “Alguma parte da tarefa foi particularmente difícil? Por quê?” . .	54

Lista de Abreviaturas e Siglas

ADC	Analog-to-Digital Converter (Conversor Analógico para Digital)
CPU	Central Processing Unit (Unidade Central de Processamento)
DC	Direct Current (Corrente Contínua)
DRAM	Dynamic Random Access Memory (Memória de Acesso Aleatório Dinâmica)
EEPROM	Electrically Erasable and Programmable Read Only Memory (Memória Somente de Leitura Programável Eletricamente Apagável)
EPROM	Erasable Programmable Read Only Memory (Memória Somente de Leitura Programável Apagável)
IDE	Integrated Development Environment (Ambiente de Desenvolvimento Integrado)
ISR	Interrupt Service Routine (Rotina de Serviço de Interrupção)
LDR	Light Dependent Resistor (Resistor Dependente de Luz)
LED	Light-emitting Diode (Diodo Emissor de Luz)
PC	Personal Computer (Computador Pessoal)
PROM	Programmable Read Only Memory (Memória Somente de Leitura Programável)
PWM	Pulse-Width Modulation (Modulação por Largura de Pulso)
RAM	Random Access Memory (Memória de Acesso Aleatório)
ROM	Read Only Memory (Memória Somente de Leitura)
SRAM	Static Random Access Memory (Memória de Acesso Aleatório Estática)
USB	Universal Serial Bus (Porta Serial Universal)

Conteúdo

Lista de Figuras	iv
Lista de Quadros	v
Lista de Abreviaturas e Siglas	vi
1 Introdução	1
1.1 Objetivos	4
1.1.1 Objetivo Geral	4
1.1.2 Objetivos Específicos	4
2 Referencial Teórico	5
2.1 Microcontroladores	5
2.1.1 Processador	5
2.1.2 Memória	6
2.1.3 Entrada e saída	7
2.1.4 Interrupções	7
2.2 Arduino	8
2.2.1 Hardware	8
2.2.1.1 Arduino UNO	10
2.2.2 Software	12
2.2.2.1 A Linguagem Arduino	13
3 Metodologia	16
3.1 Organização interna	17
3.2 Componentes passivos	18
3.2.1 Classe SinglePinDev	18
3.2.2 Classe AnalogInPinDev	19
3.2.3 Classe DigitalInSinglePin	19
3.2.4 Classe DigitalOutSinglePin	19
3.2.5 Classe Timeout	19
3.3 Componentes ativos	20
3.3.1 Classe Processable	21
3.3.1.1 Método process()	21
3.3.1.2 Autorregistro	22
3.3.1.3 Estado habilitado/desabilitado	23
3.3.1.4 Método processInstances()	23
3.3.2 Classe DigitalOutput	24
3.3.3 Classe Led	26

3.3.4	Classe Motor	28
3.3.5	Classe AnalogInputDevice	29
3.3.6	Classe Button	31
3.3.7	Classe Keypad	32
3.3.8	Classe Task	35
3.3.9	Classe Actions	36
4	Experimento	40
4.1	Introdução ao Arduino	41
4.2	Utilização da Biblioteca ArduEasy	42
4.3	Utilização da Biblioteca (API) Padrão do Arduino	42
4.4	Desafio	42
4.5	Questionários	45
4.5.1	Avaliação da biblioteca ArduEasy	45
4.5.2	Avaliação da API padrão do Arduino	46
4.5.3	Questionário comparativo	46
5	Resultados e Discussões	48
5.1	Avaliação da biblioteca ArduEasy	48
5.2	Avaliação da API padrão do Arduino	51
5.2.1	Alguma parte da tarefa foi particularmente difícil? Por quê?	54
5.3	Avaliação comparativa entre as abordagens	54
6	Conclusão	56
A	Slides do minicurso	61
A.1	Introdução ao Arduino	62
A.2	Biblioteca ArduEasy	67
A.3	Biblioteca padrão do Arduino	70

1

Introdução

Nos últimos anos, a plataforma Arduino se tornou uma das ferramentas de prototipagem eletrônica mais populares e acessíveis para estudantes de diversas áreas, hobistas, artistas e profissionais. Com origem na Itália, ganhou destaque mundial devido à sua simplicidade, versatilidade e uma vasta comunidade.

Seu sucesso pode ser atribuído a uma série de fatores inter-relacionados. Primeiramente, sua abordagem *open-source* tem sido primordial para a sua popularização. Ao disponibilizar o seu *hardware* e *software* de forma livre, a plataforma permitiu a colaboração de milhares de desenvolvedores por todo o globo.

Além disso, a plataforma é bastante acessível. Seja em relação ao custo (é possível encontrar modelos compatíveis custando a partir de R\$ 40,00) ou em relação ao seu uso facilitado quando comparado à programação de microcontroladores, graças à sua biblioteca de abstração de *hardware*, que permite a sua manipulação dispensando o conhecimento de detalhes do seu microcontrolador, seu *bootloader*, parte do *firmware* que permite a gravação do programa na placa Arduino através de sua porta *USB* ou porta Serial sem a necessidade de um *hardware* dedicado, e a sua IDE, que fornece um ambiente de desenvolvimento intuitivo e com funcionalidades que auxiliam no desenvolvimento, como autocompletar trechos de código, gerenciador de bibliotecas, *debugger*, monitor de comunicação serial, entre outros.

Embora a biblioteca do próprio Arduino implemente uma camada de abstração de *hardware*, ainda é necessária uma curva de aprendizado para iniciantes, pois precisam aprender sua linguagem de programação (C/C++ por padrão), conceitos básicos de eletrônica e a utilizar a biblioteca padrão da plataforma.

Além disso, o controle de tempo pode se tornar um desafio à medida que o projeto cresce. Em muitos casos é necessário aguardar períodos de tempo entre a leitura e escrita de sinais digitais e analógicos e, apesar da biblioteca do Arduino fornecer uma função (*delay()*) que pausa a execução do programa por um tempo determinado, esta função é bloqueante, ou seja, interrompe a execução do código até que tenha passado o tempo definido como parâmetro da

função; logo, sua utilização impactará na execução de todo o programa. A seguir será demonstrado como a simples ação de piscar LEDs pode se tornar complexa para iniciantes.

O Exemplo 1.1 mostra como piscar um LED conectado ao pino 13 do Arduino em intervalos de 500ms:

```
1 #define LED_PIN 13
2
3 byte ledState = LOW;
4
5 void setup() {
6     pinMode(LED_PIN, OUTPUT);
7 }
8
9 void loop() {
10     digitalWrite(LED_PIN, ledState);
11     ledState = !ledState;
12     delay(500);
13 }
```

Exemplo 1.1: Código para piscar LED. Fonte: De autoria própria.

Agora, supomos que se queira piscar dois LEDs, um conectado ao pino 13 em intervalos de 500ms e outro conectado ao pino 11 em intervalos de 200ms, de maneira ingênua pode-se tentar escrever o código da seguinte forma, como mostra-se no Exemplo 1.2:

```
1 #define LED_1_PIN 13
2 #define LED_2_PIN 11
3
4 byte led1State = LOW;
5 byte led2State = LOW;
6
7 void setup() {
8     pinMode(LED_1_PIN, OUTPUT);
9     pinMode(LED_2_PIN, OUTPUT);
10 }
11
12 void loop() {
13     digitalWrite(LED_1_PIN, led1State);
14     led1State = !led1State;
15     delay(500);
16
17     digitalWrite(LED_2_PIN, led2State);
18     led2State = !led2State;
19     delay(200);
20 }
```

Exemplo 1.2: Código para piscar dois LEDs implementado incorretamente de forma intencional. Fonte: De autoria própria.

Embora o código do Exemplo 1.2 realmente faça com que dois LEDs pisquem, seus intervalos de tempo não serão os desejados pois a presença das duas invocações à função *delay()* afeta o comportamento de ambos os LEDs, que agora passam a piscar a cada 700ms.

O Exemplo 1.3 implementa a proposta do Exemplo 1.2 da forma correta.

```
1 #define LED_1_PIN 13
2 #define LED_2_PIN 11
3 #define LED_1_INTERVAL 500
4 #define LED_2_INTERVAL 200
5
6 byte led1State = LOW;
7 byte led2State = LOW;
8
9 long lastLed1Timeout = 0;
10 long lastLed2Timeout = 0;
11
12 void setup() {
13     pinMode(LED_1_PIN, OUTPUT);
14     pinMode(LED_2_PIN, OUTPUT);
15 }
16
17 void loop() {
18     if (millis() - lastLed1Timeout >= LED_1_INTERVAL) {
19         lastLed1Timeout = millis();
20         led1State = !led1State;
21         digitalWrite(LED_1_PIN, led1State);
22     }
23
24     if (millis() - lastLed2Timeout >= LED_2_INTERVAL) {
25         lastLed2Timeout = millis();
26         led2State = !led2State;
27         digitalWrite(LED_2_PIN, led2State);
28     }
29 }
```

Exemplo 1.3: Código para piscar dois LEDs. Fonte: De autoria própria.

Como se pode notar, o Exemplo 1.3 não utiliza a função *delay()* e passa a utilizar a função *millis()*, que retorna a quantidade de tempo em milissegundos que a placa Arduino está em funcionamento, e a partir daí é possível armazenar quando ocorreu e obter quanto tempo se passou desde a última inversão dos estados de cada LED. Para isso, duas novas variáveis foram introduzidas *lastLed1Timeout* e *lastLed2Timeout* além de duas estruturas de decisão, uma para cada LED, aumentando a complexidade de uma simples implementação com o objetivo de fazer com que dois LEDs pisquem em intervalos diferentes para iniciantes, a qual poderia ser implementada como mostra-se no Exemplo 1.4, caso existisse uma classe *Led* que abstraísse a manipulação de um LED:

```
1 Led led1 (13);  
2 Led led2 (11);  
3  
4 void setup () {  
5     led1.blink (500);  
6     led2.blink (200);  
7 }
```

Exemplo 1.4: Piscar dois leds de forma mais intuitiva. Fonte: De autoria própria.

Diante disso, a existência de uma biblioteca que abstraia ao máximo a manipulação de sinais digitais, como leitura/escrita de sinais *HIGH* e *LOW*, permitindo a utilização de *hardwares* básicos apenas indicando o(s) pino(s) ao qual o dispositivo está conectado e definindo as ações a serem tomadas de acordo com a ocorrência de algum evento. Bem como que permita definir tarefas que devam ser executadas em intervalos de tempo pré-definidos e também a criação de ações sequenciais de forma não bloqueante poderá contribuir ainda mais para a popularidade da plataforma Arduino, facilitando o desenvolvimento de novos projetos e alcançando um público ainda maior. Uma vez que possibilitará que o seu utilizador direcione boa parte seus esforços na resolução do problema ao qual seu projeto se propõe, invés de aprender a manipular dispositivos e controlar o tempo de forma a não causar impactos na execução do seu código.

1.1 Objetivos

1.1.1 Objetivo Geral

Este trabalho visa o desenvolvimento de uma biblioteca de código aberto e de fácil uso para a plataforma Arduino, com o foco na abstração da manipulação de *hardwares* básicos como LEDs, botões, motores DC e teclados matriciais e no auxílio do controle de tempo e execução não bloqueante.

1.1.2 Objetivos Específicos

- Implementar a biblioteca utilizando o paradigma da programação orientada à objetos;
- Disponibilizar a biblioteca em um repositório público e no gerenciador de bibliotecas do Arduino.

Referencial Teórico

2.1 Microcontroladores

Microcontroladores são microcomputadores com sua memória e dispositivos de entrada e saída em um único chip, estão presentes em praticamente todos os sistemas embarcados e geralmente são compostos pelos seguintes componentes: processador (CPU), memória ROM, memória RAM e interfaces de entrada e saída, que serão detalhados em seguida (CADY, 2009).

2.1.1 Processador

O processador é parte primordial de qualquer microcontrolador e contém a unidade lógica e aritmética, que, como o próprio nome sugere, é responsável pela execução das operações lógicas e aritméticas, registradores utilizados para armazenar dados de propósito geral, como valores a serem utilizados nas operações, endereço de memória que contém a próxima instrução a ser executada, informações sobre o estado da última operação executada pelo processador, interfaces de entrada e saída e barramentos (PASQUALE, 2024).

As principais arquiteturas quanto ao acesso à memória são **Harvard**, onde a memória contendo as instruções do programa e a memória de dados são entidades separadas, como mostrado na Figura 2.1, permitindo o acesso a ambas as memórias de forma simultânea, e a arquitetura **von Neumann** em que instruções e dados compartilham a mesma memória, como mostrado na Figura 2.2, consequentemente sendo menos performática que a arquitetura Harvard.

Devido ao fato de as memórias RAM e ROM serem externas ao processador nos PCs (computadores pessoais) a arquitetura von Neumann é extensamente empregada nestes, pois o uso de memórias dedicadas para dados e programa e trilhas elétricas separadas para cada tipo de memória na placa-mãe seria bastante complexo e financeiramente mais caro (PASQUALE, 2024). Microcontroladores, por outro lado, geralmente se baseiam na arquitetura Harvard pela necessidade de gerenciar sinais digitais mais rapidamente (SCHNEIDER, 2024).

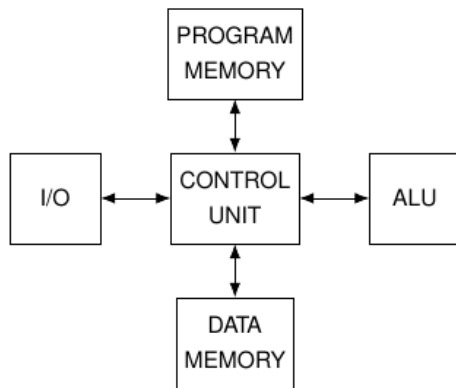


Figura 2.1: Arquitetura Harvard
Fonte: Pasquale (2024).

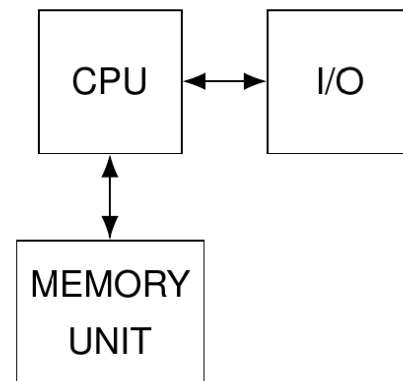


Figura 2.2: Arquitetura von Neumann
Fonte: Pasquale (2024).

2.1.2 Memória

Pode-se classificar as memórias utilizadas em microcontroladores da seguinte maneira:

- Memória volátil

Também conhecida como memória RAM (memória de acesso aleatório), é capaz de armazenar os dados somente quando energizada e são divididas em duas categorias: Static RAM (SRAM), utiliza *flip-flops* para armazenar cada bit de informação e Dynamic RAM (DRAM), que por sua vez utiliza capacitores para armazenar um bit de informação (GRIDLING; WEISS, 2007).

- Memória não volátil

Também conhecida como memória ROM (memória somente de leitura), permitem o armazenamento dos dados mesmo quando não há energia e por isso são utilizadas para o armazenamento das instruções e valores constantes nos microcontroladores (PASQUALE, 2024). Suas principais categorias são (GRIDLING; WEISS, 2007):

- PROM (ROM programável): Memória ROM que pode ser gravada/programada apenas uma vez.
- EPROM (ROM programável apagável): Memória ROM que pode ser apagada, após expor o seu chip de silício a uma luz UV e posteriormente reprogramada.
- EEPROM (ROM apagável e programável eletricamente): Memória ROM que permite que cada endereço de memória possa ser apagado e reprogramado aproximadamente 100.000 vezes. É frequentemente utilizada para o armazenamento persistente de dados.
- Flash: Uma variante da EEPROM que permite apagar blocos ou toda a memória. É mais barata, porém permite menos regravações que a EEPROM (cerca de 1.000 à

10.000 vezes) e por isso é frequentemente utilizada para armazenar o programa.

2.1.3 Entrada e saída

Uma das principais funcionalidades dos computadores de uma forma geral é a capacidade de interagir com o mundo externo através de dispositivos de entrada e saída. Nos microcontroladores isto é possível através de pinos de entrada e saída, ao qual a sua configuração (se será utilizado para entrada ou saída de dados) bem como a leitura e envio de dados estão associados à registradores (PASQUALE, 2024). Esta abordagem faz com que as operações de E/S se assemelhem com leituras e escritas em variáveis (VALVANO, 2014).

Pode-se classificar a entrada e saída em (GRIDLING; WEISS, 2007):

- Entrada digital: Basicamente é composta por dois estados, *high* e *low* que são interpretados de acordo com a tensão aplicada nos pinos correspondentes. O microcontrolador ATmega16 por exemplo, considera *low* tensões entre -0,5V e 20% da tensão de alimentação, e, considera *high* as tensões entre 60% da tensão de alimentação e 0,5V a mais que a tensão de alimentação.
- Saída digital: Utiliza-se dos estados *high* e *low* para definir a tensão disponibilizada nos pinos correspondentes. Um ATmega16 alimentado com 5V terá uma tensão máxima de 0,7V em um pino de saída quando seu estado for *low* e no máximo 4,2V quando for *high*.
- Entrada analógica: Permite que tensões aplicadas à um pino de entrada sejam convertidas em um sinal digital através de um ADC (conversor analógico para digital), possibilitando não apenas a leitura de dois estados (*high* e *low*), mas valores proporcionais à tensão aplicada no pino de entrada.
- Saída analógica: De maneira contrária à entrada analógica, a saída analógica permite que a tensão produzida no pino não seja mais restrita à valores considerados *high* ou *low* e possa sofrer oscilações. Uma forma de se atingir este objetivo é através do PWM (Modulação por largura de pulso), em que a tensão oscila entre *high* e *low* em um intervalo de tempo proporcional à tensão que se deseja produzir no pino.

2.1.4 Interrupções

Microcontroladores frequentemente necessitam reagir a mudanças de estados do sistema e isso pode ser feito de duas maneiras. De forma simples, é possível fazer verificações periódicas via *software* (*polling*) de um sinal de entrada e detectar quando o seu estado é alterado, porém essa abordagem acaba desperdiçando tempo do processador e eventos podem ser perdidos caso ocorram antes da checagem ser feita (GRIDLING; WEISS, 2007).

Outra maneira é através de interrupções, onde o próprio microcontrolador se encarrega de verificar as mudanças de estado fazendo amostragem do sinal de entrada a cada ciclo de *clock*, e

interrompe o programa principal e invoca uma rotina de serviço de interrupção (ISR) fornecida pelo programador quando uma mudança é detectada (GRIDLING; WEISS, 2007).

2.2 Arduino

Arduino é uma plataforma *open-source* projetada para artistas, designers ou outras pessoas que queiram utilizar computação física em seus projetos mesmo não sendo engenheiros eletrônicos. Permite a criação de dispositivos interativos que podem funcionar de forma autônoma ou em colaboração com algum *software* no computador (BANZI; SHILOH, 2014).

A plataforma é composta por duas partes principais: a placa Arduino, que é o *hardware* a ser utilizado para o desenvolvimento do projeto e a IDE, *software* utilizado para criar o programa que será executado na placa Arduino, também conhecido como *sketch* (BANZI; SHILOH, 2014).

2.2.1 Hardware

A placa Arduino é composta por um microcontrolador que funciona como um mini computador em um *chip* e também por componentes eletrônicos responsáveis pelo funcionamento do microcontrolador e por permitirem que a placa se comunique com um computador (BANZI; SHILOH, 2014).

Para que a placa interaja com o mundo externo, é necessária a utilização de componentes eletrônicos conhecidos como sensores e atuadores. Os sensores são capazes de converter fenômenos físicos como a quantidade de luz ambiente (Figura 2.3) e temperatura (Figura 2.4) em sinais elétricos que possam ser processados pelo microcontrolador, e a depender do valor lido, o microcontrolador pode reagir acionando algum atuador, como um LED (Figura 2.5) ou um motor elétrico (Figura 2.6) (BANZI; SHILOH, 2014).

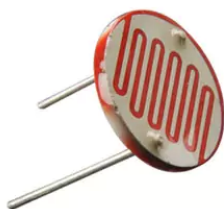


Figura 2.3: Sensor de luminosidade LDR Fonte: [tech-Zero \(2024\)](#).

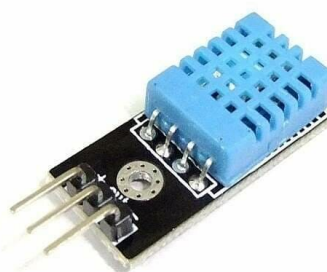


Figura 2.4: Sensor de temperatura e umidade DHT11 Fonte: [Lobo da Robótica \(2024\)](#).

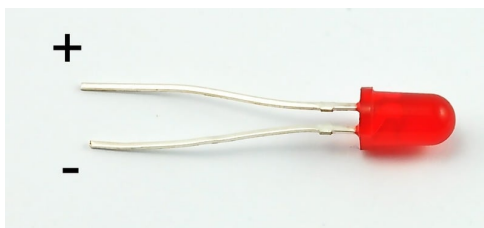


Figura 2.5: Led e seu polo positivo (+) e negativo (-) Fonte: [Adafruit Learning System \(2012\)](#).



Figura 2.6: Motor de corrente contínua Fonte: [ElectronicWings \(2024\)](#).

A placa Arduino por si só é limitada ao controle de entrada e saída e de alguns sensores e módulos de comunicação que estão presentes em alguns modelos, sendo necessário a construção de circuitos mais complexos para atender a necessidade de determinados projetos. *Shields* e módulos são circuitos pré-prontos projetados para uma função específica, logo, permitem a extensão das funcionalidades do Arduino apenas os conectando aos pinos da placa, sem que o seu utilizador necessite construir tal circuito ([SARAFAN, 2024](#)).

Shields são placas de circuitos que adicionam funcionalidades específicas ao Arduino e podem ser encaixadas diretamente a ele. A vantagem da utilização de um *shield* é que são projetados especificamente para uma versão da placa Arduino, não necessitam de fios para conectá-los e geralmente possuem uma biblioteca para o seu controle ([SARAFAN, 2024](#)).

Na Figura 2.7 mostra-se o *shield* Adafruit Motor Shield v2.3, para controle de até 4 motores DC ou até 2 motores de passo e compatível com as placas Arduino UNO, Leonardo, ADK/Mega R3, Diecimila e Duemilanove ([ADAFRUIT LEARNING SYSTEM, 2013](#)).

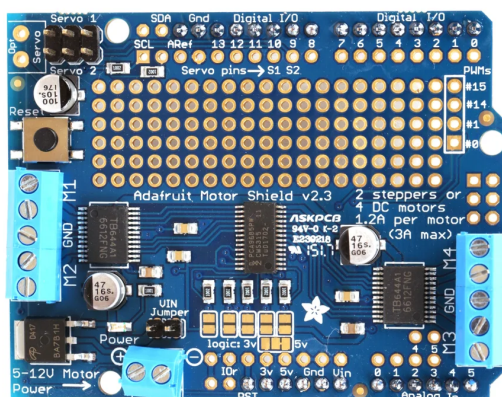


Figura 2.7: Motor *Shield*, utilizado para o controle de motores DC Fonte: [Sarafan \(2024\)](#).

Assim como os *shields* os módulos também adicionam uma funcionalidade específica ao Arduino, porém não são projetados exclusivamente para o Arduino, podendo ser utilizado por diferentes placas controladoras e alguns modelos podem até funcionar de forma autônoma.

A vantagem dos módulos é que uma vez que não precisam ser encaixados no Arduino, uma quantidade mínima de pinos será necessária para a sua conexão. Seu tamanho reduzido também permite que o projeto como um todo seja mais compacto e geralmente são menos custosos em comparação com os *shields* (SARAFAN, 2024).

Na Figura 2.8 mostra-se um módulo para controle de até dois motores DC ou um único motor de passo, equipado com o *chip* DRV8833 (SARAFAN, 2024).

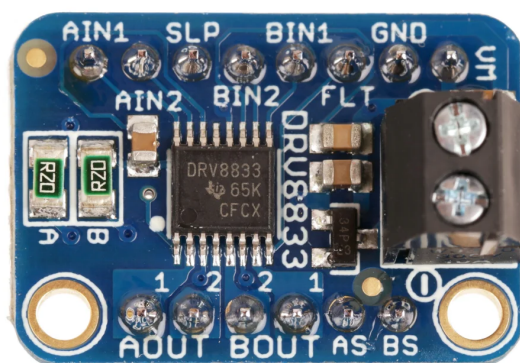


Figura 2.8: Módulo controlador de motor, utilizado para o controle de motores DC Fonte: Sarafan (2024).

Existem diversas famílias e versões de placas Arduino, a serem listadas a seguir (ARDUINO HARDWARE, 2024):

- Nano: Família composta por um conjunto de placas pequenas e alguns sensores embutidos como temperatura, umidade, pressão, gestos, microfone, entre outros. Há modelos equipados inclusive com comunicação Bluetooth e Wi-Fi.
- MKR: Família composta por placas e *shields* que podem ser combinados para a criação de projetos sem nenhum circuito adicional. Todas as placas (com exceção da MKR Zero) são equipadas com um módulo de rádio que possibilita a comunicação por Wi-Fi, bluetooth, LoRa, Sigfox e NB-IoT e utilizam o microcontrolador SAMD21.
- Classic: Família de placas que fizeram e fazem sucesso, consideradas como a espinha dorsal do projeto Arduino, como Arduino UNO, Leonardo e Micro.
- Mega: Família de placas para projetos que necessitam de mais poder computacional e pinos adicionais.

2.2.1.1 Arduino UNO

O Arduino UNO é a placa mais usada e documentada de toda a família Arduino. Baseado no microcontrolador ATmega328P, possui 14 pinos para entrada e saída digital (dos quais 6 podem ser usados como saídas PWM), 6 pinos de entrada analógica, um ressonador cerâmico de 16

MHz, uma conexão USB, um conector de alimentação elétrica, e um botão de *reset* (ARDUINO DOCUMENTATION, 2024c). Na Figura 2.9 mostra-se a placa Arduino Uno e seus principais componentes.

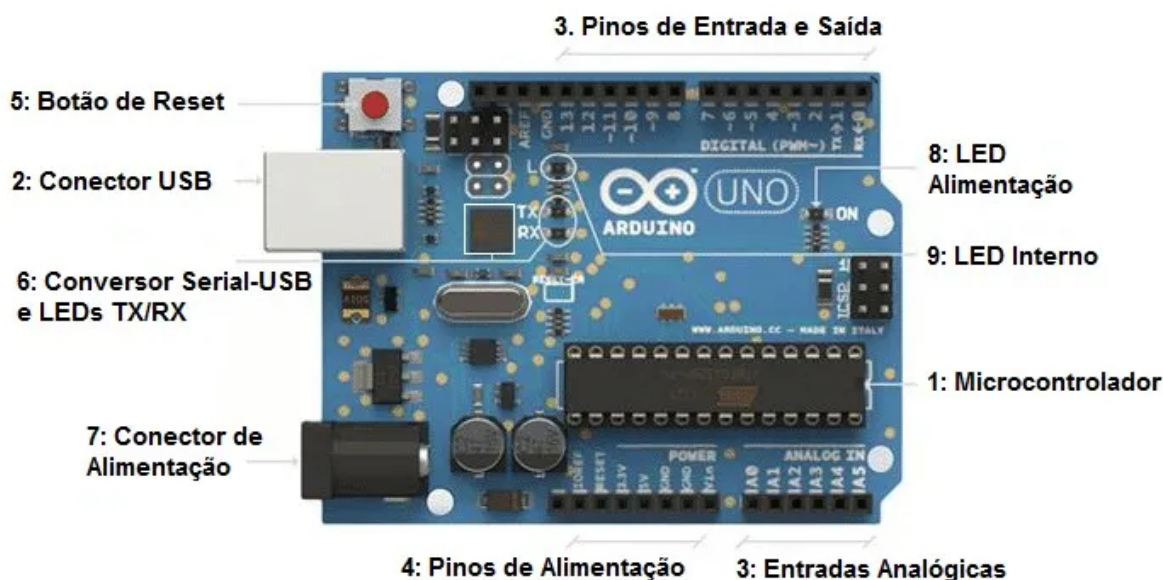


Figura 2.9: Placa Arduino UNO e seus principais componentes Fonte: Mota (2021).

O ATmega328P é um microcontrolador 8-bit baseado na arquitetura AVR. Possui uma memória Flash de 32KB, memória EEPROM de 1024B, 2KB de memória SRAM, 23 pinos de entrada e saída de uso geral e 32 registradores de uso geral e opera com tensões entre 1,8V a 5,5V (MICROCHIP, 2024).

Seus pinos de entrada e saída geralmente são organizados em grupos de oito pinos chamados de porta, cada porta é nomeada com uma letra do alfabeto, por exemplo, PortA, PortB, etc, e os pinos da porta PortA, por exemplo, são referenciados como PA0, PA1,..., PA7 (MICROCHIP, 2023). Cada porta possui três registradores associados:

- DDRx: Registrador de direção dos dados.
- PORTx: Registrador de saída.
- PINx: Registrador de entrada.

Onde “x” é o identificador da porta (A, B, C ou D) (MICROCHIP, 2023). Os *bits* de cada registrador podem ser acessados da seguinte forma:

- DDRxn: *bits* do endereço DDRx, quando escrito com “1” o pino equivalente é configurado como saída e quando escrito com “0” é configurado como entrada.

- PORT_{xn}: *bits* do endereço PORT_x e possui duas funções de acordo com a configuração do pino (entrada ou saída):
 - Saída: Quando escrito “1” o pino terá o estado *high* e *low* quando escrito “0”.
 - Entrada: Quando escrito “1” o resistor de *pull-up*¹ conectado ao pino é ativado e desativado quando escrito “0”.
- PIN_{xn}: *bits* do endereço PIN_x, utilizado para a leitura de pinos configurados como entrada. Caso o pino esteja configurado como saída, é possível ler o valor escrito através do *bit* PORT_{xn}.

Onde “x” é o identificador da porta e “n” o *bit* correspondente ao pino no registrador da porta (MICROCHIP, 2023).

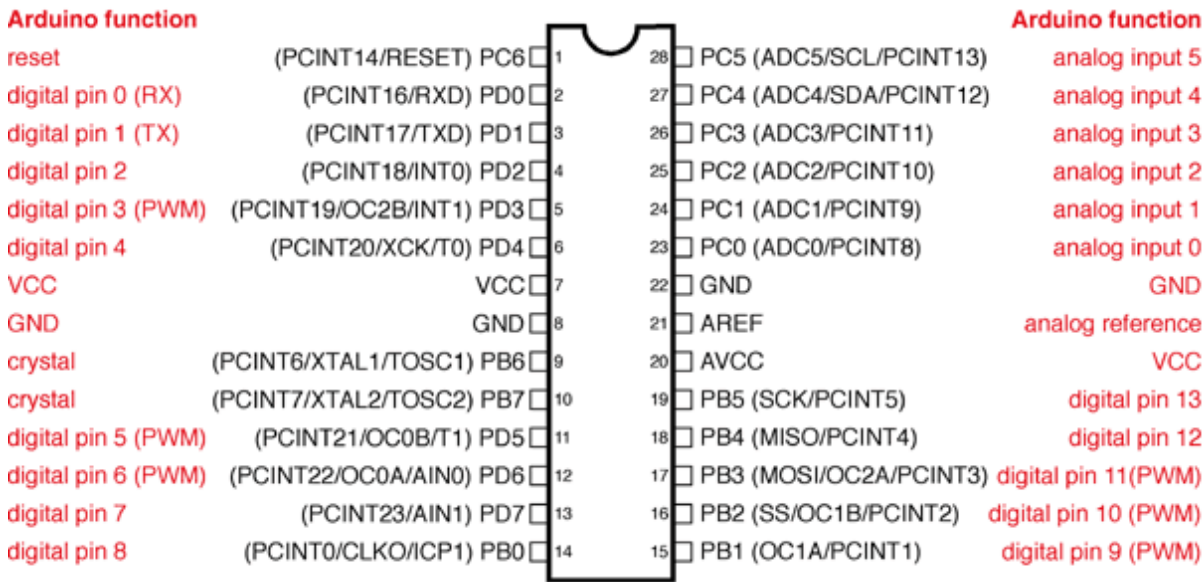


Figura 2.10: Pinagem do ATmega328P e a função correspondente no Arduino UNO Fonte: Modificado de (ARDUINO DOCUMENTATION, 2024a)

Na Figura 2.10 na cor preta temos os pinos do microcontrolador ATmega328P enumerados de 1 a 26 e seu respectivo nome/função e na cor vermelha a sua função no Arduino UNO.

2.2.2 Software

Software é uma parte crucial da plataforma Arduino, é utilizada para que o *hardware* possa interagir com o ambiente externo e também para o desenvolvimento dos *sketchs* (programa executado no Arduino) que, por sua vez, são combinados com as bibliotecas Arduino e, por fim, compilados em instruções de máquina para serem executados no microcontrolador. (WHEAT, 2011)

¹Resistor utilizado para definir o estado padrão do pino como *high* quando o mesmo estiver desconectado. (GUPTA, 2018)

A IDE Arduino é o *software* que é executado no computador e possibilita que o usuário escreva seus *sketches* (BANZI; SHILOH, 2014). Além do seu editor de código, oferece algumas funcionalidades como: autocompletar trechos de código, um gestor de placas, responsável por gerenciar os pacotes de placas disponíveis na IDE (cada pacote contém instruções de como compilar o código para a placa em questão), um gestor de bibliotecas, um monitor de comunicação serial, entre outras (SÖDERBY; HYLÉN, 2024). Segundo Banzi e Shiloh (2014) o ciclo de programação no Arduino é:

1. Conectar a placa Arduino à uma porta USB do computador.
2. Escrever um *sketch* que dará vida à placa
3. Carregar o *sketch* na placa através da conexão USB e aguardar alguns segundos até a placa reinicializar.
4. A placa executa o *sketch* programado.

Um código Arduino deverá conter duas funções chamadas *setup()* e *loop()*. A função *setup()* é onde se define todo o código que deve ser executado uma única vez na inicialização do programa, e a função *loop()* é a que contém o *core* do programa, sendo executada repetidamente (BANZI; SHILOH, 2014).

2.2.2.1 A Linguagem Arduino

O termo Linguagem Arduino é na verdade uma padronização para programação embarcada nas mais diversas arquiteturas e *hardwares* da plataforma Arduino. É uma linguagem de domínio específico implementada em C++ e que tem como objetivo prover uma abstração total de todas as chamadas de baixo nível necessária pelos *hardwares* suportados, padronizando a programação e permitindo a criação de conteúdo e exemplos que possam ser facilmente portados para outros *hardwares*. A Linguagem Arduino é composta por (RANELLUCCI, 2022):

- A API *core*: Conjunto de classes, funções e constantes que todas as implementações de uma plataforma devem fornecer.
- As bibliotecas oficiais: Fornecem abstrações dos conceitos, dispositivos e tipos de dados mais usados.

Dentre as funções definidas pela Linguagem Arduino pode-se citar por exemplo as funções *pinMode()*, para configurar um pino como entrada (*INPUT*), entrada com o resistor *pull-up* ativo (*INPUT_PULLUP*) ou saída (*OUTPUT*), *digitalRead()* para obter a tensão presente (*HIGH* ou *LOW*) em um pino definido como entrada e *digitalWrite()*, para definir a tensão presente em um pino configurado como saída (DUNBAR, 2020).

A API *core* do Arduino é apenas uma abstração e define um contrato que cada plataforma de *hardware* deverá seguir, a função *pinMode()* por exemplo é implementada da seguinte forma para o microcontrolador ATmega328P, como mostra-se no Exemplo 2.1 (DUNBAR, 2020):

```

1 void pinMode(uint8_t pin, uint8_t mode)
2 {
3     uint8_t bit = digitalPinToBitMask(pin);           ①
4     uint8_t port = digitalPinToPort(pin);             ②
5     volatile uint8_t *reg, *out;
6
7     if (port == NOT_A_PIN) return;
8
9     // JWS: can I let the optimizer do this?
10    reg = portModeRegister(port);                      ③
11    out = portOutputRegister(port);
12
13    if (mode == INPUT) {                               ④
14        uint8_t oldSREG = SREG;
15        cli();
16        *reg &= ~bit;
17        *out &= ~bit;
18        SREG = oldSREG;
19
20    } else if (mode == INPUT_PULLUP) {                 ⑤
21        uint8_t oldSREG = SREG;
22        cli();
23        *reg &= ~bit;
24        *out |= bit;
25        SREG = oldSREG;
26    } else {                                           ⑥
27        uint8_t oldSREG = SREG;
28        cli();
29        *reg |= bit;
30        SREG = oldSREG;
31    }
32 }
```

Exemplo 2.1: Implementação da função *pinMode*. Fonte: Modificado de Dunbar (2020, p. 80–81).

1. A chamada *digitalPinToBitMask()* converte o inteiro referente a um pino do Arduino para o *bitmask* equivalente na porta relacionada. Conforme a Figura 2.10, o pino digital 2 do Arduino Uno corresponde ao *bit* 2 da PortD (PD2), logo, para o pino digital 2 a macro *digitalPinToBitMask()* retornaria o *bitmask* 0000 0100.
2. A chamada *digitalPinToPort()* retorna um índice utilizado internamente na implementa-

ção do Arduino referente à porta do microcontrolador ATmega328P associada ao pino.

3. As chamadas *portModeRegister()* e *portOutputRegister()* retornam ponteiros para os registradores de direção de dados DDRx e de saída PORTx e os armazena nas variáveis *reg* e *out* respectivamente, para o pino 2 do Arduino por exemplo, *reg* irá conter o endereço de memória de DDRD e *out* o endereço de PORTD.
4. Caso o modo desejado seja entrada (*INPUT*), o *bit* associado ao pino no registrador de direção de dados (DDR) será definido como '0' (**reg &= ~bit*), e o registrador de saída (PORT) também terá seu *bit* definido como '0' (**out &= ~bit*), indicando que o resistor *pull-up* estará desativado.
5. Se o modo desejado seja entrada com resistor *pull-up* (*INPUT_PULLUP*), o *bit* do registrador de direção de dados será definido como '0' (**reg &= ~bit*) e o *bit* do registrador de saída será definido como '1' (**out |= bit*), indicando que o resistor *pull-up* estará ativo.
6. Caso o modo desejado não seja *INPUT* ou *INPUT_PULLUP* o registrador de direção de dados será definido como saída (**reg |= bit*).

3

Metodologia

A biblioteca ArduEasy permite ao seu utilizador uma forma fácil e intuitiva de controlar LEDs, botões, motores DC, teclados matriciais, entrada analógica e no controle de tempo de forma não bloqueante. Para o seu desenvolvimento optou-se por utilizar apenas a *API Core* do Arduino, sem nenhuma dependência com outras bibliotecas e sem a utilização de código específico para uma arquitetura de *hardware*, permitindo assim que *sketchs* que utilizem a biblioteca possam ser executados em qualquer família de *hardware*.

Pensou-se no desenvolvimento de abstrações como base para os componentes da biblioteca, possibilitando assim que futuramente novos componentes possam ser adicionados sem a necessidade de uma alteração estrutural na biblioteca, e, possibilitando também que usuários mais avançados possam criar seus próprios componentes para interagirem com algum dispositivo específico, por exemplo, usufruindo dos benefícios que a biblioteca proporciona. Pensou-se também em classificar os componentes como ativos, componentes que são gerenciados pela biblioteca, e, componentes passivos, que funcionam através de chamadas explícitas.

Mostra-se na Figura 3.1 como a ArduEasy está inserida no contexto de uma aplicação Arduino.



Figura 3.1: Diagrama de camadas. Fonte: De autoria própria.

Conforme mostrado na Figura 3.1, um *sketch* poderá utilizar outras bibliotecas, a API do Arduino e código específico do *hardware*, além da biblioteca ArduEasy. Cada camada poderá interagir diretamente com as camadas inferiores, com exceção da ArduEasy que visa a não dependência com outras bibliotecas e código específico de variantes do Arduino.

3.1 Organização interna

Para uma melhor organização e entendimento, as classes da biblioteca foram agrupadas em pacotes, conforme as suas responsabilidades. Mostra-se no diagrama da Figura 3.2 como foi feita tal organização, seus pacotes e como se relacionam.

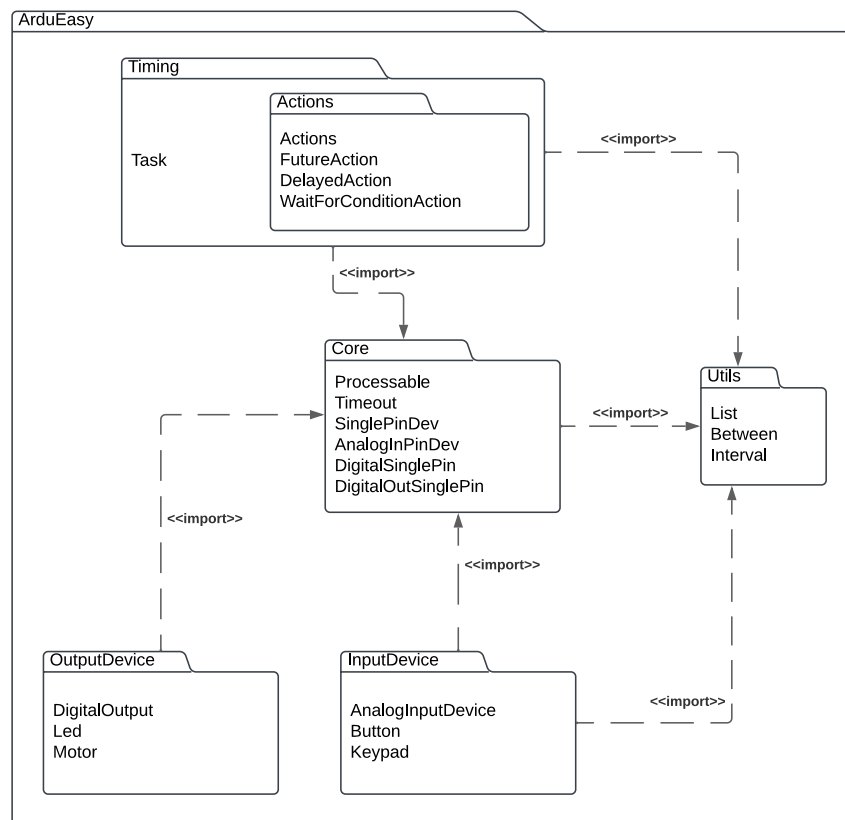


Figura 3.2: Diagrama de pacotes. Fonte: De autoria própria.

Cada pacote contém classes responsáveis por:

- **Utils:** Contém classes utilitárias que não possuem responsabilidades diretamente ligadas ao domínio da biblioteca, como uma estrutura de lista encadeada e classes utilizadas para armazenar um intervalo e verificar se um valor está contido no mesmo.
- **Core:** Contém as principais abstrações da biblioteca, servindo de base para a implementação dos componentes ativos.

- **Timing:** Contém os componentes ativos responsáveis pelo controle de tempo de forma não bloqueante.
- **Actions:** Contém classes que permitem a definição de uma sequência de ações não bloqueantes cujo o tempo de conclusão de cada ação pode ser conhecido ou não.
- **OutputDevice:** Contém classes que representam dispositivos de saída, como LEDs e motores.
- **InputDevice:** Contém classes que representam dispositivos de entrada, como botões e teclados matriciais.

3.2 Componentes passivos

São os componentes que não funcionam de forma autônoma e necessitam de chamadas explícitas de seus métodos. Via de regra são componentes utilizados para dar suporte aos componentes ativos, mas, também podem ser utilizados pelo usuário final da biblioteca, reutilizando a lógica implementada caso necessite.

Mostra-se na Figura 3.4 o diagrama de classes dos componentes passivos.

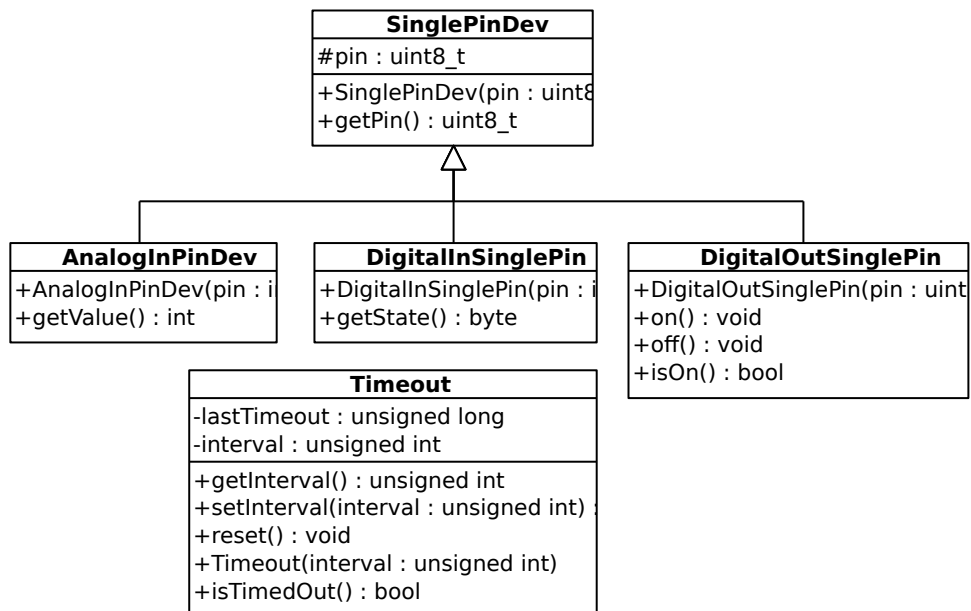


Figura 3.3: Diagrama de classes dos componentes passivos. Fonte: De autoria própria.

3.2.1 Classe SinglePinDev

Classe utilizada como base para representar dispositivos que necessitam de apenas um pino. Possui apenas o atributo *pin* que deverá ser informado no seu construtor e seu método assessor `getPin()`.

3.2.2 Classe AnalogInPinDev

Classe filha de *SinglePinDev* que representa uma entrada analógica. O pino de entrada é especificado em seu construtor, onde será configurado como entrada (*INPUT*) através da função *pinMode()* do Arduino. Possui também o método *getValue()*, responsável por fazer a leitura do pino associado utilizando a função *analogRead()* do Arduino.

3.2.3 Classe DigitalInSinglePin

Classe filha de *SinglePinDev* que representa uma entrada digital. O pino de entrada é especificado em seu construtor, onde será configurado como entrada (*INPUT*) através da função *pinMode()* do Arduino. Possui também o método *getState()*, responsável por fazer a leitura do pino associado utilizando a função *digitalRead()* do Arduino.

3.2.4 Classe DigitalOutSinglePin

Classe filha de *SinglePinDev* que representa uma saída digital. O pino de saída é especificado em seu construtor, onde será configurado como saída (*OUTPUT*) através da função *pinMode()* do Arduino. Possui os seguintes métodos:

- *on()*: Define o valor do pino de saída como *HIGH* utilizando a função *digitalWrite()* do Arduino.
- *off()*: Define o valor do pino de saída como *LOW* utilizando a função *digitalWrite()* do Arduino.
- *isOn()*: Retorna *true* caso o pino de saída esteja configurado como *HIGH*, caso contrário, retorna *false*.

3.2.5 Classe Timeout

Classe responsável por verificar se um determinado tempo já se passou. Internamente utiliza a função *millis()* do Arduino, que retorna a quantidade de tempo em milissegundos em que o programa está sendo executado. A quantidade de tempo em que ocorrerá o estouro deverá ser informada no seu construtor em milissegundos, e possui dois atributos:

- *lastTimeout*: Armazena o instante (em milissegundos) que ocorreu o último estouro de tempo.
- *interval*: Armazena a quantidade de milissegundos em que deverá acontecer o estouro de tempo.

Possui os seguintes métodos:

- *getInterval()*: Retorna o intervalo armazenado em *interval*.
- *setInterval()*: Modifica o valor armazenado em *interval*.
- *reset()*: Define o valor armazenado em *lastTimeout* como sendo o instante atual (*lastTimeout = millis()*).
- *isTimedOut()*: Verifica e retorna se houve um estouro de tempo (*millis() > lastTimeout + interval*). Em caso positivo, haverá um *reset* no valor de *lastTimeout* e será retornado *true*, em caso negativo, será retornado *false*.

Dessa forma a classe *Timeout* pode ser utilizada para o controle de tempo de forma não bloqueante, conforme o Exemplo 3.1:

```
1 //...
2 Timeout timeout(500);           ①
3
4 void loop() {
5     if(timeout.isTimedOut()) {   ②
6         acoes();                 ③
7     }
8 }
```

Exemplo 3.1: Exemplo de uso da classe *Timeout*. Fonte: De autoria própria.

1. Cria uma instância de *Timeout* chamada *timeout*, com intervalo de 500ms.
2. Verifica se houve o estouro de tempo.
3. Caso tenha ocorrido o estouro de tempo, executa as ações definidas no bloco de código.

3.3 Componentes ativos

São todos os componentes que, de certa forma, são controlados pela biblioteca, funcionando de forma autônoma, cabendo ao programador apenas configurá-los, informando o pino em que um LED está conectado, por exemplo, e definir algumas ações através de *callbacks*, como uma função a ser executada quando um botão for pressionado.

Mostra-se na Figura 3.4 o diagrama de classes dos componentes ativos.

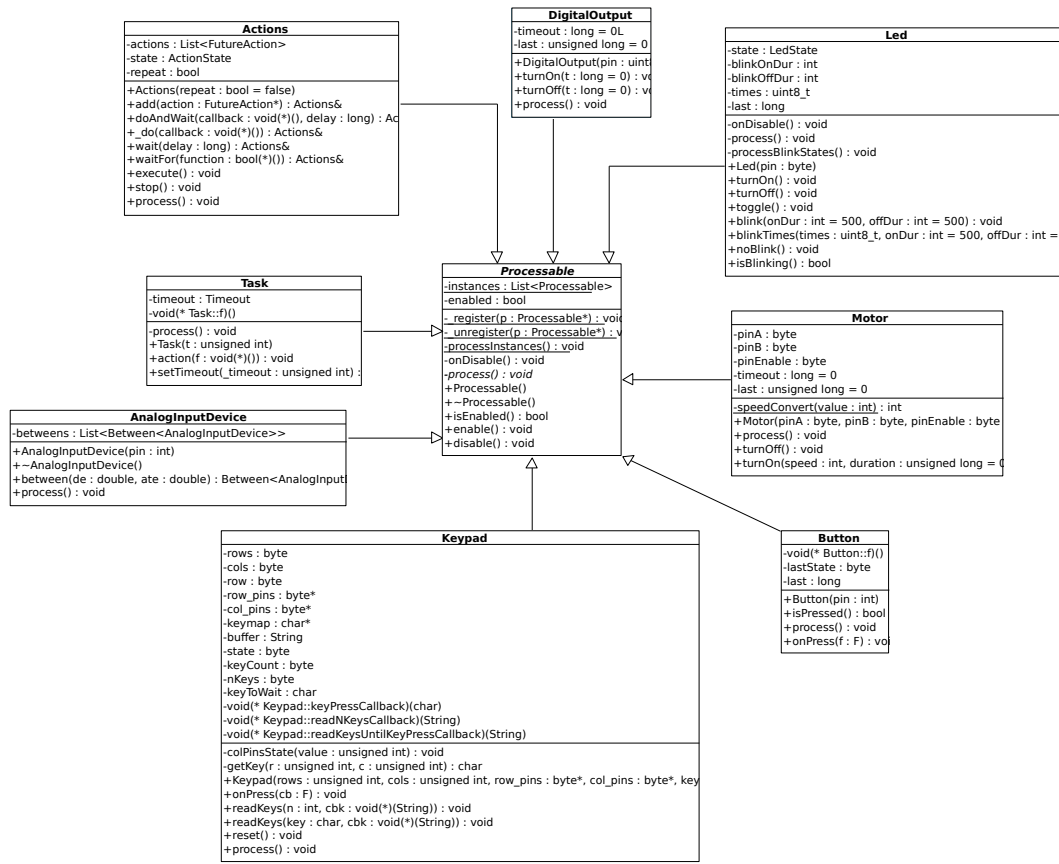


Figura 3.4: Diagrama de classes dos componentes ativos. Fonte: De autoria própria.

3.3.1 Classe Processable

A espinha dorsal dos componentes ativos é a classe abstrata *Processable*, utilizada como base para todos os componentes que necessitam de uma execução periódica, seja para o controle de um *hardware*, como verificar se um botão foi pressionado, por exemplo, ou execuções puramente de *software*, como executar uma ação após um determinado tempo. Para obter essa característica “ativa” necessitou-se de:

1. A declaração de um método abstrato para que as classes filhas implementem a sua lógica.
2. Um mecanismo de autorregistro, possibilitando que a biblioteca possa acessar todas as instâncias de *Processable*.

3.3.1.1 Método process()

É onde cada implementação de *Processable* definirá a sua lógica de funcionamento. Uma vez que a biblioteca será responsável pela invocação deste método, cada implementação deverá se responsabilizar pelo seu controle de estado. O pseudo-código representado pelo Exemplo

3.2 exemplifica como seria uma implementação de *Processable* responsável por fazer um LED piscar a cada 200ms.

```
1 class BlinkLed : Processable {  
2     //...  
3     void process() {  
4         if (past200ms()) {           ①  
5             resetTimeout();          ②  
6             if (!isOn()) {           ③  
7                 on();                 ④  
8             } else {  
9                 off();                ⑤  
10            }  
11        }  
12    }  
13 }
```

Exemplo 3.2: Exemplo de uma implementação da classe *Processable* para piscar um LED.
Fonte: De autoria própria.

1. Verifica se já se passaram 200ms desde que o LED ligou ou desligou e caso contrário nada será feito até que *process* seja invocado novamente.
2. Após ter passado 200ms, o controle de tempo da implementação será resetado.
3. Verifica se o estado atual do LED não é *on*, executando o item 4 caso a condição seja satisfeita, ou o item 5 caso a condição não seja satisfeita.
4. Altera o estado do LED para *on*.
5. Altera o estado do LED para *off*.

Conforme o Exemplo 3.2, não importa quantas vezes o método *process* seja invocado, a implementação garante que nada será feito caso não tenha passado 200ms desde a última alteração do estado do LED.

3.3.1.2 Autorregistro

A classe *Processable* possui um mecanismo de "autorregistro" com a finalidade de tornar possível que a biblioteca tenha acesso a suas instâncias. Automaticamente, o endereço de memória de cada instância é inserido ou removido em uma lista privada estática à medida em que uma instância de uma classe filha é criada ou destruída.

3.3.1.3 Estado habilitado/desabilitado

Processable possui uma *flag* que indica se essa instância está habilitada ou não. Isto permite que o programador desative algum componente temporariamente ou sob alguma condição que julgue adequada. Na prática, a biblioteca verificará o estado desta *flag* antes de invocar o método *process*.

Em alguns casos, seria interessante que alguma medida fosse tomada sempre que uma implementação seja desabilitada. No Exemplo 3.2, o LED ficaria sempre aceso caso a instância seja desabilitada no momento em que seu estado seja *on*. Pensando nisso, *Processable* disponibiliza o método *onDisable()* que pode ser sobrescrito por suas implementações para que adicionem toda a lógica do comportamento esperado quando se é desabilitado. O pseudocódigo representado pelo Exemplo 3.3 mostra como seria implementado o método *onDisable* para sempre desligar o LED quando *BlinkLed* for desativada.

```
1 class BlinkLed : Processable {  
2     //...  
3     void onDisable() {  
4         off();  
5     }  
6 }
```

Exemplo 3.3: Exemplo de uma implementação de *onDisable*. Fonte: De autoria própria.

3.3.1.4 Método *processInstances()*

Para de fato dar “vida” às instâncias de *Processable* pensou-se na criação de um método estático privado chamado *processInstances()* com o seguinte comportamento:

1. Acessar todas as instâncias de *Processable*.
2. Invocar o método *process()* de todas as instâncias que estão ativas.

O trecho de código do Exemplo 3.4 demonstra o seu funcionamento.

```
1 class Processable {  
2     //...  
3 private:  
4     void processInstances() {  
5         instances.forEach([](Processable &p) {  
6             if (p.enabled) {  
7                 p.process();  
8             }  
9         });  
10    }  
11 }
```

Exemplo 3.4: Implementação do método *processInstances()*. Fonte: De autoria própria.

Esta abordagem, em que de tempos em tempos é permitido que as instâncias executem suas tarefas se assemelha com o escalonamento cooperativo de tarefas, e, dá a impressão de que as instâncias estão executando suas ações de forma simultânea, uma vez que é possível reagir ao clique de um botão ao mesmo tempo em que um LED está piscando, por exemplo, porém, caso uma implementação consuma muito tempo na execução do seu método *process*, afetará o comportamento de toda a biblioteca.

Uma vez que este método é privado, disponibilizou-se a função *run()* sob o *namespace* *ArduEasy* conforme o trecho de código do Exemplo 3.5, para que o programador possa invocar a execução das instâncias de *Processable*.

```
1 void ArduEasy::run() {  
2     Processable::processInstances();  
3 }
```

Exemplo 3.5: Implementação da função *ArduEasy::run()*. Fonte: De autoria própria.

Conforme os Exemplos 3.4 e 3.5, uma invocação da função *ArduEasy::run()* percorrerá a lista de instâncias uma única vez, sendo insuficiente para que as instâncias funcionem de forma ativamente. Para isso é necessário que o método *ArduEasy::run()* seja invocado repetidamente, e uma das formas de invocá-lo dessa maneira é através da função *loop()* do Arduino.

O trecho de código representado pelo Exemplo 3.6 mostra a implementação de um *sketch* Arduino utilizando a biblioteca *ArduEasy*.

```
1 #include<ArduEasy.h>  
2  
3 // Declaração das variáveis  
4  
5 void setup() {  
6     // Configuração dos componentes  
7 }  
8  
9 void loop() {  
10     ArduEasy::run();  
11 }
```

Exemplo 3.6: Implementação de um sketch do Arduino com a biblioteca *ArduEasy*. Fonte: De autoria própria.

3.3.2 Classe *DigitalOutput*

Classe filha de *Processable* e *DigitalOutSinglePin* que, além de abstrair a utilização de um pino como saída digital, também permite temporizar o seu estado como *HIGH* ou *LOW*. Pode ser utilizada para acionamento de relés, por exemplo. O pino a ser controlado deverá ser informado no seu construtor, e possui os seguintes atributos:

- *timeout*: Armazena a quantidade de tempo em milissegundos em que ocorrerá o estouro de tempo. Quando seu valor é igual 0, indica que a instância não está atuando no modo "temporizado". Quando positivo, indica que está no modo temporizado e que o estado do pino deverá ser definido como *LOW* após o estouro de tempo. Quando negativo, indica que também está no modo temporizado, porém o estado do pino deverá ser definido como *HIGH* após o estouro de tempo.
- *last*: Armazena o instante em que a instância passou a atuar no modo temporizado.

Optou-se por não utilizar a classe `Timeout` já que o atributo *timeout* também define (através do seu sinal) qual será o estado do pino após o estouro de tempo, caso contrário, seria necessário incluir um outro atributo para indicar qual seria o estado do pino após o estouro de tempo, aumentando assim a quantidade de memória necessária.

Mostra-se no trecho de código do Exemplo 3.7 a implementação dos seus métodos *turnOn*, *turnOff* e *process*.

```
1  void turnOn(long t = 0) {
2      on();
3      if (t) {
4          timeout = t;
5          last = millis();
6      }
7  }
8
9  void turnOff (long t = 0) {
10     off();
11     if (t) {
12         timeout = -t;
13         last = millis();
14     }
15 }
16
17 void process() {
18     if (timeout != 0 && (millis() - last > abs(timeout))) {
19         timeout > 0 ? turnOff() : turnOn();
20     }
21 }
```

Exemplo 3.7: Implementação dos métodos *turnOn*, *turnOff* e *processable* da classe *DigitalOutput*. Fonte: De autoria própria.

Conforme o trecho de código do Exemplo 3.7, o método *turnOn* inicialmente define o estado do pino como *HIGH* ao invocar o método *on*, define o valor de *timeout* e inicializa o atributo *last* caso seu argumento *t* seja diferente de 0. O método *turnOff* funciona de maneira semelhante a *turnOn*, porém inicialmente define o pino como *LOW* ao invocar o método *off* e, quando o

argumento t é diferente de 0, define *timeout* como $-t$. Já no método *process* é verificado se está no modo temporizado ($timeout \neq 0$) e se houve o estouro de tempo, e, em caso positivo, altera o estado do pino.

3.3.3 Classe Led

Classe filha de [Processable](#) e [DigitalOutSinglePin](#) responsável pelo controle de LEDs. Permite ligar, desligar e piscar LEDs de forma temporizada ou não. Em seu construtor deverá ser informado o pino ao qual o LED está conectado e possui os seguintes atributos:

- *state*: *Enum* que representa o estado atual do LED e seus possíveis valores são:
 - OFF: Indica que o LED está sempre desligado.
 - ON: Indica que o LED está sempre ligado.
 - BLINKING: Indica que o LED está piscando.
 - BLINKING_TIMES: Indica que o LED está piscando por um determinado número de vezes.
- *blinkingOnDur*: Indica a quantidade de milissegundos em que o LED ficará ligado quando seu estado for *BLINKING* ou *BLINKING_TIMES*.
- *blinkingOffDur*: Indica a quantidade de milissegundos em que o LED ficará desligado quando seu estado for *BLINKING* ou *BLINKING_TIMES*.
- *times*: Indica a quantidade de vezes que o LED piscará quando seu estado for *BLINKING_TIMES*.
- *last*: Armazena o instante em que o LED ligou ou desligou quando o seu estado for *BLINKING* ou *BLINKING_TIMES*.

Os principais métodos da classe Led são:

- *turnOn()*: Define o estado do LED como *ON*.
- *turnOff()*: Define o estado do LED como *OFF*.
- *toggle()*: Alterna entre os estados do LED. Quando o estado atual do LED for:
 - *ON*: Define o estado do LED como *OFF*.
 - *OFF*: Define o estado do LED como *ON*.
 - *BLINKING* ou *BLINKING_TIMES*: Não altera o estado do LED.

- *blink(int onDur = 500 , int offDur = 500)*: Define o estado do LED como *BLINKING*, os argumentos *onDur* e *offDur* indicam o tempo em milissegundos em que o LED ficará ligado e desligado, respectivamente, ambos possuem 500 milissegundos como valor padrão.
- *blinkTimes(uint8_t times, int onDur = 500 , int offDur = 500)*: Define o estado do LED como *BLINKING_TIMES*, o argumento *times* indica a quantidade de vezes que o LED deverá piscar, já os argumentos *onDur* e *offDur* indicam o tempo em milissegundos em que o LED ficará ligado e desligado, respectivamente, ambos possuem 500 milissegundos como valor padrão.
- *noBlink()*: Assim como o método *turnOff()*, define o estado do LED como *OFF*.

Mostra-se no trecho de código do Exemplo 3.8 a implementação dos métodos *process* e *processBlinkStates*

```
1 void process() {
2     if (state == OFF) {
3         off();
4     } else if (state == ON) {
5         on();
6     } else {
7         processBlinkStates();
8     }
9 }
10
11 void processBlinkStates() {
12     int s = isOn();
13     int duration = s ? blinkOnDur : blinkOffDur;
14     long now = millis();
15     if (now > last + duration) {
16         last = now;
17
18         if (isOn()) {
19             off();
20         } else {
21             if (state == BLINKING_TIMES) {
22                 if (times > 0)
23                     times--;
24                 else {
25                     state = OFF;
26                     return;
27                 }
28             }
29             on();
30 }
```

```
31 |     }  
32 | }
```

Exemplo 3.8: Implementação dos métodos `process` e `processBlinkStates`. Fonte: De autoria própria.

Conforme o trecho de código do Exemplo 3.8, no método `process()` o LED é desligado quando seu estado é *OFF*, ligado quando seu estado é *ON* e quando seu estado é *BLINKING* ou *BLINKING_TIMES* o método `processBlinkStates` é invocado. O método `processBlinkStates` por sua vez decide se utilizará o valor de `blinkOnDur` ou `blinkOffDur` para verificar se houve um estouro de tempo com base no estado atual do pino do LED e inverte o estado do pino em caso positivo. Quando há o estouro e o estado do LED for *BLINKING_TIMES* e o LED estiver desligado, o valor de `times` é decrementado enquanto for maior que 0 ou o estado do LED é alterado para *OFF* quando `times` for igual a 0, fazendo com que o LED pare de piscar.

O fato de o comportamento da classe LED ser determinado pelo seu estado atual a torna candidata à aplicação do padrão de projeto *State*, porém optou-se pela não utilização deste padrão uma vez que necessitaria de objetos representando tais estados e, conseqüentemente, ocuparia mais espaço em memória, recurso este escasso em sistemas embarcados como o Arduino.

3.3.4 Classe Motor

Classe filha de `Processable` responsável pelo controle de motores de corrente contínua. Deve ser usada juntamente com *drivers* para controle de motores presentes em módulos ou *shields*, como o *driver* DRV8833 presente no módulo da Figura 2.8. A grande maioria dos *shields* e módulos necessitam de 3 pinos digitais para o controle de motores, 2 destes determinam o sentido de rotação do motor, para isso, ambos deverão ter estados diferentes, ou seja, quando um for *LOW* o outro deverá ser *HIGH* e essa combinação determinará o seu sentido de rotação, e um pino *PWM* para determinar a velocidade do motor.

Seu construtor requer os seguintes argumentos: `pinA` e `pinB`, ambos os pinos que determinarão o sentido de rotação, e o `pinEnable`, pino *PWM* que irá definir a velocidade do motor. Todos os pinos são definidos como saída (*OUTPUT*) através da função do Arduino `pinMode`, o estado de `pinA` é definido inicialmente como *HIGH* e `pinB` e `pinEnable` como *LOW*, utilizando a função `digitalWrite` do Arduino.

A classe possui os seguintes atributos:

- `pinA` e `pinB`: Pinos usados para definir o sentido de rotação do motor.
- `pinEnable`: Pino utilizado para definir a velocidade do motor.
- `timeout`: Objeto do tipo `Timeout` utilizado para controlar, caso necessário, o tempo em que o motor deve permanecer ligado de forma não bloqueante.

Possui os seguintes métodos:

- *turnOn(int speed, unsigned long duration = 0)*: Liga o motor por um tempo determinado ou indefinido. Seus argumentos são:
 - *speed*: Utilizado para indicar tanto a velocidade como sentido de rotação. Seu valor deverá indicar em porcentagem a velocidade do motor, ou seja, com valores de 0 a 100 o motor a velocidade irá variar entre estar parado até a sua velocidade máxima. Já o seu sinal determinará o sentido de rotação do motor, ou seja, *turnOn(100)* fará o motor girar na velocidade máxima em um sentido, e, *turnOn(-100)* fará o motor girar na velocidade máxima no sentido contrário.
 - *duration*: Indica a quantidade de tempo em milissegundos em que o motor deverá permanecer ligado, quando seu valor for 0 ou for omitido, o motor permanecerá ligado por tempo indefinido.
- *turnOff()*: Desliga o motor e define o intervalo de *timeout* como 0.
- *speedConvert(int value)*: Método privado que faz a conversão de porcentagem para o respectivo valor a ser utilizado no pino *PWM* definido em *pinEnable* através da função do Arduino *analogWrite*.

O seu método *process* apenas verifica se a instância está atuando no modo de tempo controlado (*timeout.getInterval() != 0*) e, caso esteja, verifica se houve um estouro de tempo, desligando o motor em caso positivo, conforme mostra o trecho de código do Exemplo 3.9.

```
1 void process() {  
2     if (timeout.getInterval() != 0 && timeout.isTimedOut()) {  
3         turnOff();  
4     }  
5 }
```

Exemplo 3.9: Implementação do método processable da classe Motor. Fonte: De autoria própria.

3.3.5 Classe AnalogInputDevice

Classe filha de [Processable](#) e [AnalogInPinDev](#) que pode ser utilizada para fazer a leitura explícita do valor analógico do pino, através do método *getValue()* de [AnalogInPinDev](#), e, também para executar ações definidas por meios de *callbacks* quando o valor analógico do pino estiver contido em um intervalo especificado. Em seu construtor deverá ser informado o pino a ser controlado pela instância e possui um único atributo *betweens*, uma lista de objetos do tipo *Between*, responsável por armazenar todos os intervalos de valores a serem verificados periodicamente e o *callback* correspondente a ser executado quando o valor analógico do pino estiver contido neste intervalo.

Esta classe implementa o método *between(double from, double to)*, que insere uma instância de *Between* na lista utilizando o intervalo definido por *from* e *to* (este sendo inclusivo) e a retorna, conforme o trecho de código do Exemplo 3.10.

```
1 Between<AnalogInputDevice> &between(double from, double to) {  
2     Between<AnalogInputDevice> *e = new Between<AnalogInputDevice>(*this,  
3         from, to);  
4     betweens.add(e);  
5     return *e;  
}
```

Exemplo 3.10: Implementação do método *between*. Fonte: De autoria própria.

Conforme o Exemplo 3.10, uma referência para a instância de *AnalogInputDevice* (**this*) é passada como primeiro argumento ao instanciar um objeto do tipo *Between*. Isso se deve ao fato de *Between* possuir um método chamado *then(void (*callback)())*, que além de armazenar o *callback* referente a este intervalo, retorna a referência da instância de *AnalogInputDevice*, permitindo assim o encadeamento de chamadas ao método *between*, facilitando a escrita e leitura do código ao se definir vários intervalos, conforme o Exemplo 3.11.

```
1 #include "ArduEasy.h"  
2  
3 AnalogDevice moistureSensor(A0);  
4  
5 void setup() {  
6     moistureSensor.between(0, 512).then(openValve)  
7         .between(513, 1024).then(closeValve);  
8 }  
9  
10 void openValve() {  
11     ...  
12 }  
13  
14 void closeValve() {  
15     ...  
16 }  
17  
18 void loop() {  
19     ArduEasy::run();  
20 }
```

Exemplo 3.11: Utilização do método *between*. Fonte: De autoria própria.

Como mostra-se no Exemplo 3.11, a possibilidade de um encadeamento de métodos disponibilizado por *Between* permite que todos os intervalos e seus *callbacks* sejam definidos em uma única declaração, sem a necessidade de repetirmos a variável *moistureSensor* para cada definição de intervalo e *callback*.

Já no método *process*, a cada execução os intervalos são testados e seus *callbacks* são invocados caso o valor analógico esteja contido no intervalo conforme o trecho de código do Exemplo 3.12.

```
1 void process() {  
2     betweens.first();  
3     while (betweens.hasNext()) {  
4         betweens.next()->testAndInvokeCallback(getValue());  
5     }  
6 }
```

Exemplo 3.12: Implementação do método *process*. Fonte: De autoria própria.

3.3.6 Classe Button

Classe filha de *DigitalInSinglePin* e *Processable* responsável pela execução de uma ação quando um botão for pressionado. Em seu construtor deverá ser especificado o pino ao qual o circuito do botão está conectado e possui os seguintes atributos:

- *callback*: Ponteiro para a função a ser executada quando o botão for pressionado.
- *lastState*: Armazena o último estado do botão. É necessário para contornar o efeito *bouncing*, em que ao pressionar um botão, oscilações do estado do pino poderão ocorrer em um curto espaço de tempo devido as características de construção do botão, fazendo com que o programa interprete como múltiplos pressionamentos quando não tratado (SILVA, 2019).
- *timeout*: Utilizado para esperar, de forma não bloqueante, um tempo predefinido (50ms por padrão) para contornar o efeito *bouncing*.

Implementa os métodos *onPress(void (*callback)())*, utilizado para definir o *callback* a ser invocado quando o botão for pressionado, e o método *process*, que utiliza a técnica de *polling* para verificar periodicamente o estado do pino. Esse método também verifica se houve o estouro de tempo a fim de contornar o efeito *bouncing* e, sempre que o estado do pino passa de *LOW* para *HIGH* e há um *callback* definido, invoca o *callback*, conforme o trecho de código do Exemplo 3.13.

```
1 void process() {  
2     if (timeout.isTimedOut()) {  
3         byte state = getState();  
4         if (state && state != lastState && callback != nullptr) {  
5             callback();  
6         }  
7         lastState = state;  
8     }
```

9 }

Exemplo 3.13: Implementação do método `process`. Fonte: De autoria própria.

A classe *Button* é uma forte candidata à adoção do padrão de projeto *Observer*, uma vez que não utilizaria um único *callback* para executar uma ação, permitindo assim a separação da lógica de ações com diferentes responsabilidades, as quais seriam notificadas sempre que o botão fosse pressionado. Porém, optou-se por não utilizar este padrão por se tratar de um sistema embarcado, sendo que tal implementação necessitaria de instâncias implementando a interface *observer* definida, ocupando mais espaço em memória.

3.3.7 Classe Keypad

Classe filha de *Processable* utilizada para leitura de teclados matriciais, tipo de teclado em que botões são organizados em linhas e colunas formando uma matriz, esta abordagem permite que menos pinos sejam necessários para detectar o pressionamento de uma tecla (PICORETI, 2017). A Figura 3.5 ilustra a ligação interna de um teclado matricial do tipo 4x4.

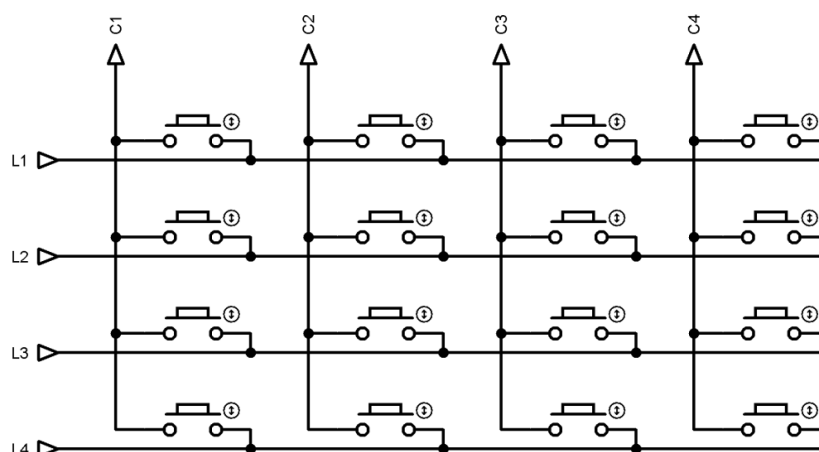


Figura 3.5: Ligação interna de um teclado matricial 4x4. Fonte: Picoreti (2017).

A classe *Keypad* permite utilizar teclados matriciais de diferentes tamanhos de linha e coluna, a especificação do mapa de caracteres que cada tecla representa, e *callbacks* que serão invocados quando qualquer tecla for pressionada, quando uma quantidade especificada de teclas for pressionada, ideal para leitura de senhas de comprimento fixo por exemplo, e quando uma tecla utilizada como indicador do fim da entrada for pressionada (como uma tecla 'enter' por exemplo). Para facilitar a sua instanciação, optou-se por implementar a classe de forma parametrizada, ou seja, fazendo uso de *templates* do C++ para indicar a quantidade de linhas e colunas disponíveis no teclado gerenciado em tempo de compilação, tornando a especificação do mapa de caracteres do teclado (um *array* de duas dimensões do tipo *char*) mais intuitiva, especialmente para iniciantes, e com melhor legibilidade. Seu construtor requer os seguintes atributos:

- *row_pins*: Array indicando quais são os pinos referentes as linhas.
- *col_pins*: Array indicando quais são os pinos referentes as colunas.
- *keymap*: Array de duas dimensões do tipo *char* representando o mapa de caracteres.

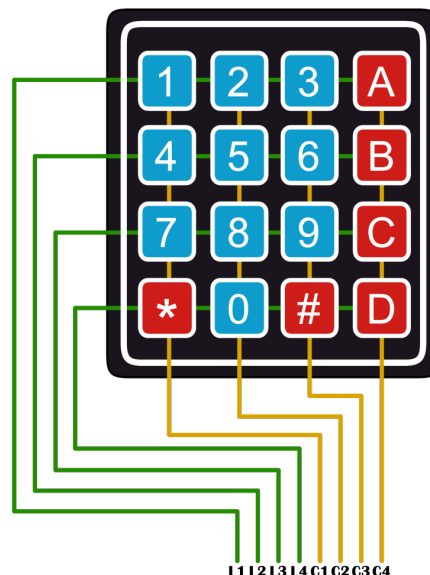


Figura 3.6: Pinagem de um teclado matricial 4x4. Fonte: [Picoreti \(2017\)](#).

A instanciação da classe *Keypad* para um teclado matricial 4x4, como ilustrado na Figura 3.6, supondo que as linhas L1 a L4 estejam conectadas aos pinos 4 a 7 do Arduino, e as colunas C1 a C4 estejam conectadas aos pinos 8 a 11 do Arduino, se dá da seguinte forma, como mostra o Exemplo 3.14:

```

1 #include "ArduEasy.h"
2
3 const byte row_pins[] = { 4, 5, 6, 7 };
4 const byte col_pins[] = { 8, 9, 10, 11 };
5 const char keypadmap[4][4] = {
6     { '1', '2', '3', 'A' },
7     { '4', '5', '6', 'B' },
8     { '7', '8', '9', 'C' },
9     { '*', '0', '#', 'D' }
10 };
11
12 Keypad<4, 4> keypad(row_pins, col_pins, keypadmap);

```

Exemplo 3.14: Instanciação da classe *Keypad* para um teclado 4x4. Fonte: De autoria própria.

A classe *Keypad* possui os seguintes atributos:

- *row*: Coluna correspondente da tecla que foi recém pressionada.

- *row_pins*: Referência para o *array* que armazena os pinos das linhas.
- *col_pins*: Referência para o *array* que armazena os pinos das colunas.
- *keymap*: Referência para o *array* de duas dimensões que armazena o mapa de caracteres.
- *buffer*: *String* utilizada para armazenar as teclas pressionadas até que os *callbacks* que necessitam desta *string* sejam invocados.
- *state*: *Enum* que representa o estado atual da instância:
 - *WAITING*: Verificando o estado dos pinos das linhas para detectar quando seu estado for alterado.
 - *PRESSED*: Detectando qual a coluna correspondente da tecla pressionada, para então obter o caractere correspondente e invocar os *callbacks* relacionados.
 - *RELEASED*: A tecla pressionada já foi liberada e por isso o estado da instância voltará a ser *WAITING*.
- *nKeys*: Indica a quantidade de teclas que devem ser pressionadas antes do *callback readNKeysCallback* ser invocado. Caso seu valor seja 0, o *callback* não será invocado.
- *keyCount*: A quantidade de teclas pressionadas desde a última execução do *callback readNKeysCallback* quando *nKeys* é diferente de 0.
- *keyToWait*: Indica o caractere que determina o fim da leitura e invoca o *callback readKeysUntilKeyPressCallback*.
- *keyPressCallback*: *Callback* que é invocado sempre que uma tecla é pressionada, passando a tecla pressionada como parâmetro.
- *readNKeysCallback*: *Callback* que é invocado passando o *buffer* como argumento sempre que o número de teclas definido por *nKeys* é pressionado.
- *readKeysUntilKeyPressCallback*: *Callback* que é invocado passando o *buffer* como argumento sempre que a tecla definida por *keyToWait* é pressionada.

Seus principais métodos são:

- *onPress(void (*cb)(char))*: Atribui o *callback* recebido como parâmetro ao atributo *keyPressCallback*, indicando o *callback*.
- *readKeys(int n, void (*cbk)(String))*: Atribui o parâmetro *n* ao atributo *nKeys* e o *callback cbk* ao atributo *readNKeysCallback*.

- `readKeys(char key, void (*cbk)(String))`: Atribui o parâmetro `key` ao atributo `keyToWait` e o `callback cbk` ao atributo `readKeysUntilKeyPressCallback`.

Seu método `process` apenas invoca outros métodos responsáveis pela lógica de cada estado, conforme o trecho de código do Exemplo 3.15:

```
1 void process() {  
2     if (state == WAITING) {  
3         processWaiting();  
4     } else if (state == PRESSED) {  
5         processPressed();  
6     } else {  
7         processReleased();  
8     }  
9 }
```

Exemplo 3.15: Implementação do método `process` da classe `Keypad`. Fonte: De autoria própria.

Assim como na classe `Button`, a adoção do padrão de projetos *Observer* também se aplicaria a `Keypad`, porém optou-se pela não adoção pelos mesmos motivos.

3.3.8 Classe Task

Classe filha de `Processable` que permite a execução de tarefas periódicas de forma não bloqueante. Em seu construtor, deverá ser informado o intervalo de tempo em milissegundos em que a tarefa deve ser executada e possui dois atributos: `timeout`, uma instância da classe `Timeout` utilizada na detecção do estouro de tempo, e `callback`, que armazena o endereço de memória da função que deverá ser executada sempre que houver um estouro de tempo. Seus principais métodos são:

- `action(void (*callback)())`: Responsável por definir o valor do atributo `callback`.
- `setTimeout(unsigned int _timeout)`: Responsável por modificar o intervalo de tempo em que deverá ocorrer o estouro de tempo.

Seu método `process` é bem simples, apenas verifica se há um `callback` definido e o invoca quando houver um estouro de tempo, conforme o trecho de código do Exemplo 3.16:

```
1 void process() {  
2     if (callback != nullptr && timeout.isTimedOut()) {  
3         callback();  
4     }  
5 }
```

Exemplo 3.16: Implementação do método `process` da classe `Task`. Fonte: De autoria própria.

3.3.9 Classe Actions

Os componentes ativos abordados até este momento resolvem os problemas que motivaram a sua construção, porém ainda não é possível criar algumas ações não bloqueantes mais complexas, como por exemplo, ao pressionar um botão aguardar N milissegundos e só então executar uma ação, pois a utilização de uma função bloqueante como a *delay()* na função de *callback* do botão irá parar a execução de todo o código durante o tempo definido na função *delay()*.

Diferentemente dos sistemas operacionais presentes em *hardwares* mais robustos, como computadores pessoais e *smartphones*, o Arduino não possui suporte nativo a *threads*. Embora seja possível implementar um escalonador de tarefas preemptivo, essa solução é complexa, exige manipulação específica do *hardware* e, por esse motivo, descartou-se esta abordagem.

Para resolver este problema, pensou-se na criação da classe *Actions*, que será responsável por definir um conjunto de ações sequenciais a serem executadas de forma não bloqueante, e, para este fim, necessitou-se da criação da classe abstrata *FutureAction*, utilizada para representar o estado e definir a lógica de uma ação cujo seu tempo de conclusão é conhecido ou não, como uma espera por um tempo pré-definido ou por algum evento externo, como a interação com uma pessoa, respectivamente.

As instâncias de *FutureAction* poderão assumir 3 estados:

- *STOPPED*: Quando está aguardando a sua execução.
- *RUNNING*: Quando está sendo executada.
- *FINISHED*: Quando sua execução terminou.

e, suas classes filhas deverão implementar o método abstrato *execute*, responsável pela definição de toda a lógica da ação, assim como a classe *Processable* e o método *process* e, opcionalmente, as implementações também poderão implementar os métodos *reset*, invocado sempre que a ação é concluída, sendo útil para a implementação redefinir os valores de seus atributos para uma próxima execução, por exemplo, e o método *onStateChange* utilizado para notificar sempre que o estado da ação for alterado. *FutureAction* também possui os métodos *terminate*, que apenas define o seu estado como *FINISHED* e os métodos *setState* e *getState* responsáveis por definir e obter o estado da ação, respectivamente.

Atualmente há apenas duas implementações primordiais de *FutureAction*, são elas *WaitForConditionAction*, que possui um *callback* cujo retorno é do tipo lógico (*boolean*) e a ação só é concluída quando o seu retorno for verdadeiro (*true*), e *PostDelayedAction* responsável por executar uma ação (*callback*) imediatamente e sua conclusão ocorrerá apenas quando houver o estouro de um tempo em milissegundos predefinido. O *callback* de *PostDelayedAction* poderá ser nulo, indicando que a ação irá apenas esperar o tempo informado, assim como o tempo informado pode ser 0, indicando que a ação irá invocar o *callback* e concluir em seguida.

Já a classe *Actions* herda *Processable* e funciona como uma orquestradora de ações que implementam *FutureAction*. Internamente, possui um estado *RUNNING* ou *STOPPED* que

indicam se as ações devem ser executadas ou não, possui um atributo *repeat* indicando se a sua execução é contínua, ou seja, deve executar todas as ações novamente após o término da última ação, e possui também uma lista de ações (instâncias de *FutureAction*) gerenciadas. Seus principais métodos são:

- *add(FutureAction *action)*: Adiciona uma ação no final da lista de ações.
- *doAndWait(void (*callback)(), long delay)*: Instancia uma ação do tipo *PostDelayedAction* e a adiciona ao final da lista, com o *callback* e tempo de espera definidos pelos argumentos *callback* e *delay*.
- *_do(void (*callback)())*: Instancia uma ação do tipo *PostDelayedAction* e a adiciona ao final da lista, com o *callback* definido pelo argumento *callback* e tempo de espera sendo 0.
- *wait(long delay)*: Instancia uma ação do tipo *PostDelayedAction* e a adiciona ao final da lista, com o *callback* sendo nulo e tempo de espera sendo definido pelo argumento *delay*.
- *waitFor(bool (*function)())*: Instancia uma ação do tipo *WaitForConditionAction* e a adiciona ao final da lista, com o *callback* definido pelo argumento *function*.
- *execute()*: Define o estado de *Actions* como *RUNNING*.
- *stop()*: Define o estado de *Actions* como *STOPPED*.

Os métodos *add*, *doAndWait*, *_do*, *wait* e *waitForCondition* retornam uma referência para a própria instância de *Actions*, possibilitando assim invocá-los de forma encadeada, melhorando a legibilidade do código.

Já o seu método *process* é responsável por todo o gerenciamento das ações, como mostra-se no trecho de código do Exemplo 3.17:

```
1 void process () {  
2     if (state != RUNNING || actions.getSize () == 0) {  
3         return; ①  
4     }  
5  
6     FutureAction *currentAction = actions.getCurr (); ②  
7  
8     if (currentAction->getState () == FINISHED) { ③  
9         currentAction->setState (STOPPED);  
10        actions.next ();  
11        if (actions.hasNext ()) {  
12            currentAction = actions.getCurr ();  
13        } else if (repeat) {  
14            currentAction = actions.first ();  
15        } else {
```

```
16         stop();
17         return;
18     }
19 }
20
21 if (currentAction->getState() == STOPPED) {           ④
22     currentAction->reset();
23     currentAction->setState(RUNNING);
24 }
25
26 if (currentAction->getState() == RUNNING) {           ⑤
27     currentAction->execute();
28 }
29 }
```

Exemplo 3.17: Implementação do método process da classe Actions. Fonte: De autoria própria.

Explicação resumida:

1. Retorna imediatamente caso o estado de *Actions* não seja *RUNNING* ou nenhuma ação foi definida ainda.
2. Obtém a ação que está sendo gerenciada no momento.
3. Verifica se a ação que está sendo gerenciada já finalizou, e em caso positivo, define seu estado como *STOPPED*, verifica se a ação concluída era a última ação, e em caso positivo, se prepara para executar a primeira ação na próxima invocação de *process* ou invoca o método *stop* caso o modo de repetição não esteja habilitado.
4. Verifica se ação que está sendo gerenciada está aguardando ser iniciada (*STOPPED*), em caso positivo, invoca seu método *reset* e altera o seu estado para *RUNNING*.
5. Verifica se o estado da ação que está sendo gerenciada é *RUNNING* e em caso positivo invoca o seu método *execute*.

Dessa forma se torna possível executar ações sequenciais de forma não bloqueante, e suponha-se que o programador deseja inverter o estado de um LED dois segundos após um botão ser pressionado e que o LED e o botão estão conectados aos pinos 13 e 2 do Arduino respectivamente, isso poderá ser facilmente implementado como mostra-se no Exemplo 3.18.

```
1 #include "ArduEasy.h"
2
3 Led led(13);
4 Button button(2);
5
6 bool buttonPressed = false;
7
```

```
8 Actions actions(true);
9
10 void resetButtonPressed() {
11     buttonPressed = false;
12 }
13
14 void setup() {
15     button.onPress([]{
16         buttonPressed = true;
17     });
18
19     actions._do(resetButtonPressed)           ①
20         .waitFor([]{return buttonPressed;})   ②
21         .wait(2000)                           ③
22         ._do([]{led.toggle();})               ④
23         .execute();                           ⑤
24 }
25
26 void loop() {
27     ArduEasy::run();
28 }
```

Exemplo 3.18: Inverter o estado de um LED dois segundos após um botão ser pressionado.
Fonte: De autoria própria.

O Exemplo 3.18 atribui *true* à variável *buttonPressed* sempre que o botão é pressionado, e cada linha de configuração da variável *actions* é responsável por:

1. De início, define *buttonPressed* como *false* ante de verificar se o botão foi pressionado.
2. Aguarda até que o botão seja pressionado.
3. Após o botão ter sido pressionado, espera por dois segundos.
4. Inverte o estado do LED.
5. Indica que este conjunto de ações deve ser executado.



Experimento

Este capítulo apresenta a metodologia adotada para a realização de um experimento, cujo objetivo foi avaliar a usabilidade e a eficiência da biblioteca ArduEasy no ensino de conceitos fundamentais de Arduino para iniciantes. Para isso, foi ministrado um minicurso para estudantes do ensino médio, abordando os seguintes temas: introdução ao Arduino, uso da biblioteca ArduEasy e utilização da API padrão do Arduino.

A avaliação foi conduzida por meio da aplicação de desafios práticos e questionários, a fim de coletar percepções sobre a dificuldade de implementação e a preferência dos participantes em relação às abordagens apresentadas.

O experimento foi realizado com nove estudantes do ensino médio, os quais foram divididos em dois grupos, e cada grupo recebeu os conteúdos na seguinte ordem:

- Grupo 1:
 1. Introdução ao Arduino
 2. Biblioteca ArduEasy
 3. Implementação do desafio utilizando a biblioteca ArduEasy
 4. Questionário sobre a biblioteca ArduEasy
 5. Biblioteca padrão do Arduino
 6. Implementação do desafio utilizando a biblioteca padrão
 7. Questionário sobre a biblioteca padrão
 8. Questionário comparativo
- Grupo 2:
 1. Introdução ao Arduino

2. Biblioteca padrão do Arduino
3. Implementação do desafio utilizando a biblioteca padrão
4. Questionário sobre a biblioteca padrão
5. Biblioteca ArduEasy
6. Implementação do desafio utilizando a biblioteca ArduEasy
7. Questionário sobre a biblioteca ArduEasy
8. Questionário comparativo

Os questionários serão detalhados na seção [4.5](#).

A divisão dos grupos teve como propósito comparar a influência da ordem de aprendizado na experiência e na percepção dos participantes em relação à facilidade de implementação dos desafios.

O minicurso foi dividido em três módulos principais, descritos a seguir.

4.1 Introdução ao Arduino

O primeiro módulo teve como objetivo familiarizar os participantes com os conceitos fundamentais do Arduino, abrangendo os seguintes tópicos:

- Histórico e relevância do Arduino no contexto da prototipagem eletrônica.
- Estrutura do hardware e pinos de conexão do Arduino.
- Definição de sensores e atuadores.
- Apresentação da *Arduino IDE*.
- Fundamentos da linguagem C++ aplicada ao Arduino.
 - Declaração e utilização de variáveis.
 - Estruturas de decisão.
 - Estruturas de repetição.
 - Definição e uso de funções.
 - A importância e utilização de bibliotecas.

Os slides deste módulo estão disponíveis na seção [A.1](#) do Apêndice [A](#).

4.2 Utilização da Biblioteca ArduEasy

O segundo módulo teve como foco a apresentação e utilização da biblioteca ArduEasy, desenvolvida no contexto deste trabalho. Foram abordados os seguintes aspectos:

- Como utilizar a biblioteca.
- Controle de LEDs.
- Leitura de botões.
- Controle de tempo não bloqueante utilizando a classe *Actions*.

Os slides deste módulo estão disponíveis na seção [A.2](#) do Apêndice [A](#).

4.3 Utilização da Biblioteca (API) Padrão do Arduino

O terceiro módulo abordou a programação utilizando exclusivamente a API padrão do Arduino. Foram apresentados os seguintes conceitos:

- Configuração de pinos com a função *pinMode*.
- Controle de saída digital por meio da função *digitalWrite*.
- Leitura de entrada digital com a função *digitalRead*.
- Manipulação de tempo utilizando as funções *delay* e *millis*.
- Implementação de controle de tempo não bloqueante.
- Aplicação prática do acionamento de LEDs e leitura de botões.

Os slides deste módulo estão disponíveis na seção [A.3](#) do Apêndice [A](#).

4.4 Desafio

Após a exposição dos conteúdos sobre cada abordagem, os participantes foram desafiados a implementar um sistema de semáforo utilizando um módulo semáforo composto por três LEDs nas cores vermelho, amarelo e verde (Figura [4.1](#)) e um módulo botão (Figura [4.2](#)) para ativação e desativação do sistema. Utilizaram também um *sensor shield* para facilitar a conexão dos módulos com o Arduino e uma placa Arduino UNO.

Os participantes receberam as seguintes instruções: quando ativo, o semáforo deveria alternar entre os LEDs na seguinte ordem e tempos de duração:

1. LED verde: 5 segundos
2. LED amarelo: 3 segundos
3. LED vermelho: 8 segundos

Após o término do LED vermelho, o ciclo deveria ser reiniciado. Quando o sistema estivesse desativado, todos os LEDs deveriam permanecer apagados.

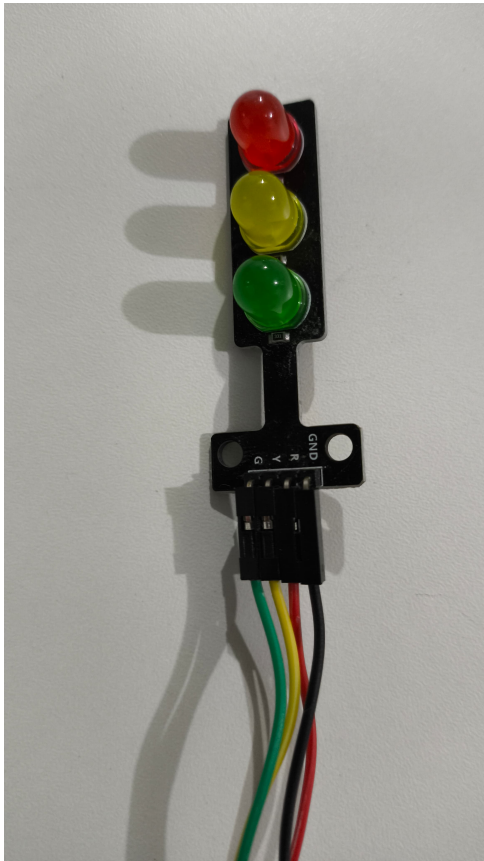


Figura 4.1: Módulo semáforo. Fonte: De autoria própria.

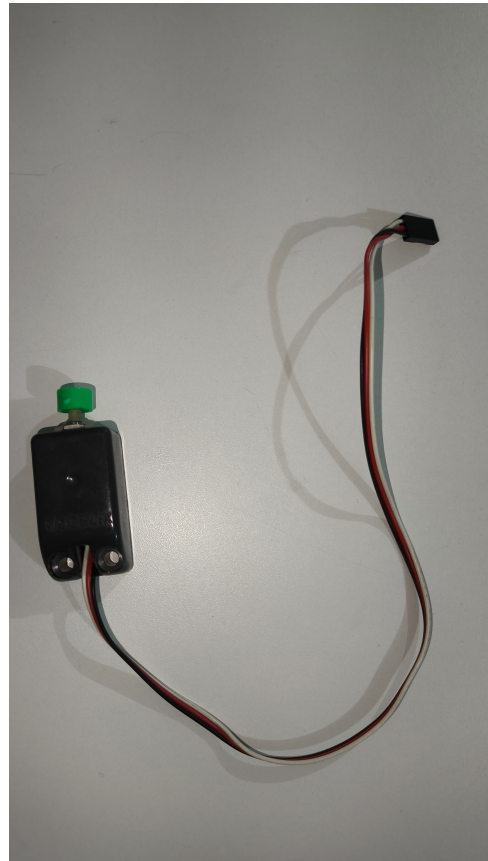


Figura 4.2: Módulo botão. Fonte: De autoria própria.

A Figura 4.3 ilustra um exemplo da montagem do protótipo utilizado para o desafio (os pinos utilizados por cada participante podem variar):

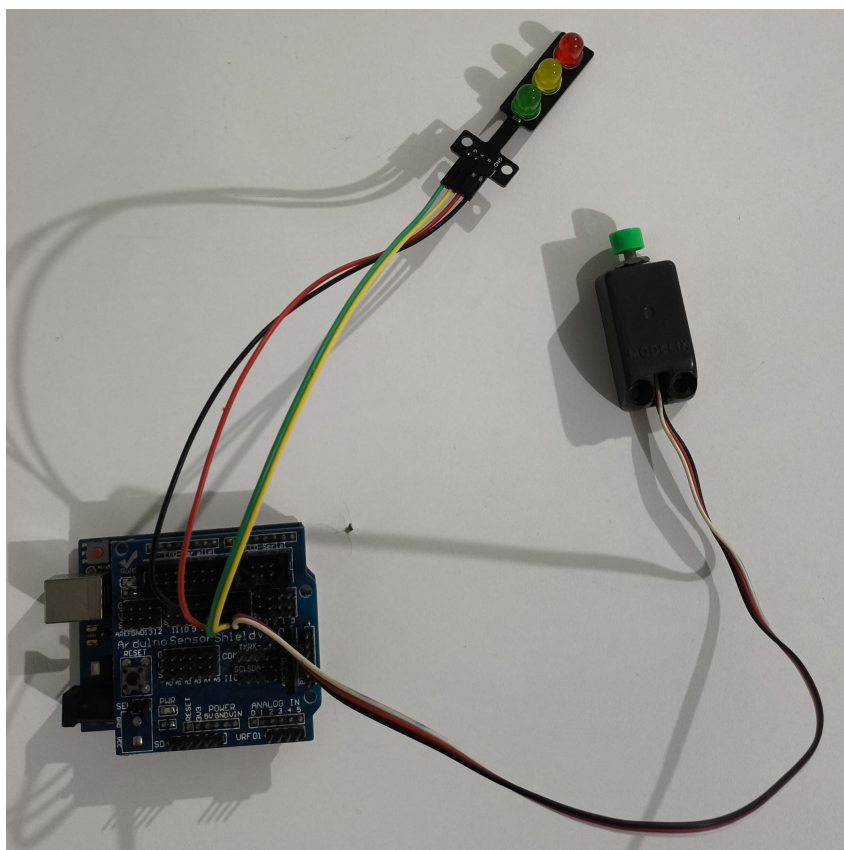


Figura 4.3: Protótipo de um semáforo. Fonte: De autoria própria.

Para garantir a avaliação comparativa das abordagens, os participantes deveriam utilizar exclusivamente a metodologia recém-apresentada para a implementação do desafio.

Para a implementação com a biblioteca *ArduEasy*, era esperado que utilizassem uma instância da classe *Led* para cada LED do semáforo e utilizassem uma instância da classe *Actions* para definir as ações de cada LED (ligar ou desligar) e o tempo de espera entre cada uma das ações, e, por fim, era esperado que utilizassem uma instância da classe *Button* cujo *callback* associado ao método *onPress* fosse responsável por interromper ou retomar a execução das ações que controlam os LEDs.

Já na implementação utilizando apenas a API padrão do Arduino, era esperado que, através da função *pinMode* definissem os pinos conectados aos LEDs como saída e o pino conectado ao botão como entrada, o uso das funções *digitalWrite* para definir o estado de cada LED e *digitalRead* para obter o estado atual do botão, bem como a utilização da função *millis* para o controle de tempo da transição de cada LED de forma não bloqueante (uma vez que o uso de *delay* afetaria a leitura do estado do botão) e também para o tratamento do efeito *bounce* do botão, além de variáveis auxiliares para o controle de tempo e para armazenar o estado do semáforo como um todo (ativado ou desativado).

Após a conclusão de cada desafio, os estudantes responderam a um questionário avaliativo, no qual relataram suas impressões quanto à dificuldade encontrada na implementação.

Ao término do minicurso, foi aplicado um questionário comparativo, no qual os participantes avaliaram as diferenças entre as abordagens e indicaram sua preferência para projetos futuros envolvendo LEDs, botões e controle de tempo não bloqueante.

4.5 Questionários

Com o objetivo de avaliar a usabilidade e a eficiência da biblioteca ArduEasy em comparação com a API padrão do Arduino, foram aplicados três questionários distintos aos participantes do experimento. Cada questionário foi elaborado para obter percepções qualitativas e quantitativas acerca das experiências durante a execução das tarefas propostas. As subseções a seguir descrevem detalhadamente o conteúdo e o propósito de cada um dos instrumentos aplicados.

4.5.1 Avaliação da biblioteca ArduEasy

Para avaliar a experiência dos participantes com a biblioteca ArduEasy, foi elaborado um questionário composto por questões quantitativas e qualitativas.

As questões quantitativas exigiam que os participantes classificassem suas percepções utilizando a seguinte escala de cinco pontos:

1. Muito difícil (Tive grande dificuldade ou não consegui realizar a tarefa)
2. Difícil (Consegui, mas com muita dificuldade)
3. Médio (Foi um desafio, mas consegui entender e implementar)
4. Fácil (Tive pouca dificuldade e consegui resolver bem)
5. Muito fácil (Foi intuitivo e consegui resolver sem dificuldades)

As questões foram organizadas da seguinte forma:

- **Questões quantitativas:**

- Quão fácil foi compreender a tarefa proposta?
- Quão fácil foi usar a biblioteca ArduEasy para resolver o desafio?
- Quão difícil foi organizar e escrever o código usando a biblioteca ArduEasy?
- Quão fácil foi entender e aplicar os métodos da ArduEasy?
- Quão fácil foi gerenciar o controle de tempo e tarefas usando a ArduEasy?

- **Questões qualitativas:**

- Qual foi o maior benefício de usar a biblioteca ArduEasy?
- Alguma parte da tarefa foi particularmente difícil? Por quê?
- Quais funcionalidades da ArduEasy você considerou mais úteis?

4.5.2 Avaliação da API padrão do Arduino

Da mesma forma, foi aplicado um questionário para avaliar o uso da API padrão do Arduino, também composto por questões quantitativas e qualitativas, com a mesma escala de avaliação mencionada anteriormente.

- **Questões quantitativas:**

- Quão fácil foi compreender a tarefa proposta?
- Quão fácil foi usar a API padrão do Arduino para resolver o desafio?
- Quão difícil foi organizar e escrever o código usando a API padrão do Arduino?
- Quão fácil foi aprender e aplicar os conceitos e métodos da API padrão?
- Quão fácil foi gerenciar o controle de tempo não bloqueante na API padrão?

- **Questões qualitativas:**

- Qual foi o maior desafio que você enfrentou ao usar a API padrão do Arduino?
- Alguma parte da tarefa foi particularmente difícil? Por quê?

4.5.3 Questionário comparativo

Com o objetivo de comparar diretamente a experiência com a biblioteca ArduEasy e com a API padrão do Arduino, foi elaborado um questionário exclusivamente com questões quantitativas. As perguntas e opções de resposta estão descritas a seguir:

- Qual abordagem foi mais fácil de entender e utilizar?
 - API padrão muito mais fácil
 - API padrão um pouco mais fácil
 - Ambas foram iguais
 - ArduEasy um pouco mais fácil
 - ArduEasy muito mais fácil
- Em qual abordagem o código ficou mais simples e legível?
 - API padrão muito mais simples
 - API padrão um pouco mais simples
 - Ambas foram iguais
 - ArduEasy um pouco mais simples
 - ArduEasy muito mais simples

- Qual abordagem permitiu resolver o problema mais rapidamente?
 - API padrão muito mais rápida
 - API padrão um pouco mais rápida
 - Ambas levaram o mesmo tempo
 - ArduEasy um pouco mais rápida
 - ArduEasy muito mais rápida
- Se você fosse iniciar um novo projeto com Arduino em que precisasse controlar o tempo sem bloqueios, e manipular LEDs e botões, de que forma faria?
 - Utilizaria apenas a API padrão
 - Utilizaria a biblioteca ArduEasy

Resultados e Discussões

A avaliação dos dados coletados por meio dos questionários permitiu analisar a efetividade da biblioteca ArduEasy em comparação à API padrão do Arduino. A seguir, são discutidos os principais resultados referentes ao objetivo deste trabalho: simplificar o uso do Arduino para iniciantes por meio de uma biblioteca fácil e intuitiva.

5.1 Avaliação da biblioteca ArduEasy

De acordo com os gráficos apresentados na Figura 5.1, os participantes demonstraram uma forte percepção positiva em relação à biblioteca ArduEasy.

Quanto à facilidade no uso da biblioteca para resolver o desafio, observa-se na terceira coluna do gráfico, referente ao conjunto completo de participantes, que 88,9% destes classificaram como “fácil” ou “muito fácil” e apenas 11,1% classificaram como “médio”.

Já, quando questionados sobre a dificuldade em organizar e escrever o código utilizando a ArduEasy, as respostas deste mesmo conjunto estiveram mais distribuídas entre as categorias, onde 33,3% classificaram como “muito fácil”, 22,2% como “fácil”, 33,3% como “médio” e 11,1% como “difícil”. Isso sugere que, apesar das facilidades fornecidas pela biblioteca, faz-se necessário um enfoque mais aprofundado na introdução à linguagem e lógica de programação comparado ao que foi apresentado no minicurso.

Quanto à facilidade em entender e aplicar os métodos, observa-se na terceira coluna do gráfico que 66,7% classificou como “fácil” ou “muito fácil” e apenas 33,3% como “médio”. Já, quando questionados sobre a facilidade em gerenciar o tempo utilizando a biblioteca, todas as respostas se concentraram entre “fácil” (66,7%) e “muito fácil” (33,3%), sugerindo que a forma abordada para o controle de tempo não bloqueante da biblioteca ArduEasy (classe Actions) é intuitiva.

Nota-se de maneira geral que as respostas do Grupo 1, referente aos participantes que foram expostos a ArduEasy primeiro, tendem a ser mais negativas que as do Grupo 2, referente aos

participantes que foram expostos a API padrão primeiro. Isso sugere que o fato de o Grupo 2 ter sido exposto primeiro a uma abordagem possivelmente mais difícil tornou as percepções sobre a biblioteca ArduEasy mais positivas.

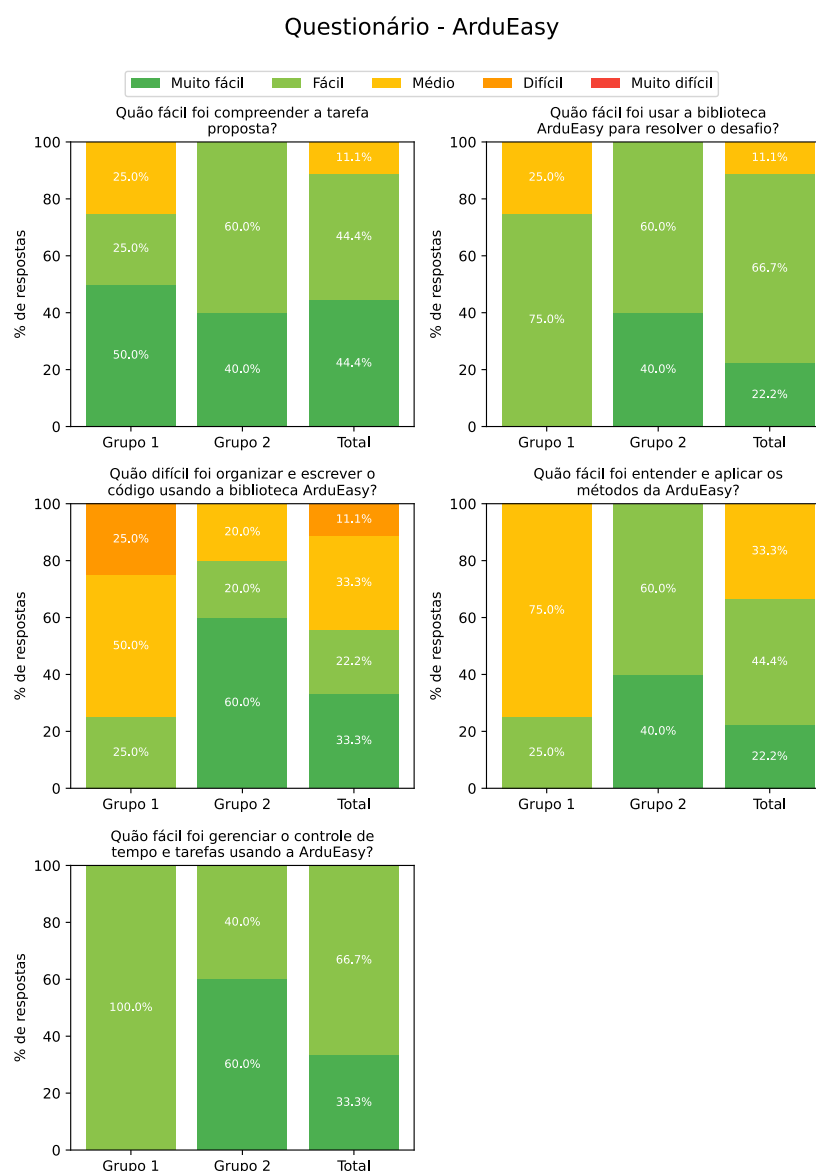


Figura 5.1: Gráficos das respostas do formulário de avaliação da ArduEasy. Fonte: De autoria própria.

As respostas qualitativas, organizadas nos Quadros 5.1 a 5.3, reforçam a percepção positiva da ArduEasy. Quando questionados sobre o maior benefício da biblioteca (Quadro 5.1), os participantes mencionaram, de forma recorrente, a praticidade (62,5%) e a simplicidade (37,5%). Em relação às dificuldades enfrentadas (Quadro 5.2), a maioria (66%) afirmou que não houve grandes obstáculos, ou mencionou aspectos mais ligados à lógica do projeto do que à biblioteca em si, o que reforça a necessidade de uma maior ênfase na introdução à linguagem e lógica de programação. Por fim, ao indicar quais funcionalidades consideraram mais úteis (Quadro 5.3),

a maioria (86%) dos participantes apontou que todas as funcionalidades vistas no minicurso foram úteis.

Qual foi o maior benefício de usar a biblioteca ArduEasy?	Grupo
Que é mais rápido	Grupo 2
Que foi mais rápido e simples	Grupo 2
O maior benefício foi a facilidade de executar o comando.	Grupo 1
Facilitar o trabalho.	Grupo 1
Pratico	Grupo 1
Pratica e facilidade na hora de digitar os códigos	Grupo 2
Facilidade	Grupo 2
Mais prático	Grupo 2

Quadro 5.1: Respostas de “Qual foi o maior benefício de usar a biblioteca ArduEasy?”. Fonte: De autoria própria.

Alguma parte da tarefa foi particularmente difícil? Por quê?	Grupo
Não	Grupo 2
não	Grupo 2
Não. A parte mais complexa foi adicionar e determinar as ações, mas nada muito difícil.	Grupo 1
Memorizar os códigos, por conta da sua complexidade.	Grupo 1
escrever o código	Grupo 1
Formular a lógica	Grupo 2
nao achei	Grupo 2
Nao	Grupo 2
nenhuma	Grupo 2

Quadro 5.2: Respostas de “Alguma parte da tarefa foi particularmente difícil? Por quê?”. Fonte: De autoria própria.

Quais funcionalidades da ArduEasy você considerou mais úteis?	Grupo
Todas foram úteis.	Grupo 1
Ser prático.	Grupo 1
todas	Grupo 1
todas	Grupo 1
Todas	Grupo 2
Todas	Grupo 2
todas	Grupo 2

Quadro 5.3: Respostas de “Quais funcionalidades da ArduEasy você considerou mais úteis?”.
Fonte: De autoria própria.

Os resultados evidenciam que a biblioteca ArduEasy foi bem recebida pelos participantes, sendo considerada prática e fácil. Isso reforça a proposta da ArduEasy de facilitar o aprendizado e a implementação de projetos com Arduino por meio de um método mais intuitivo, principalmente para iniciantes. Ao mesmo tempo, os dados sugerem a importância de dedicar maior atenção à introdução à linguagem e à lógica de programação, fundamentais para que os utilizadores aproveitem plenamente os recursos da biblioteca e possam alcançar os objetivos do seu projeto.

5.2 Avaliação da API padrão do Arduino

Por outro lado, os dados sobre a API padrão revelam maiores dificuldades. Conforme ilustrado na Figura 5.2, as tarefas foram consideradas mais complexas, especialmente no que diz respeito à organização e escrita do código e ao controle de tempo não bloqueante.

Quanto à facilidade em resolver o desafio, 55,6% dos participantes consideraram “médio”, 33,33% como “difícil” e somente 11,11% consideraram “fácil”, o que evidencia um contraste em comparação com as respostas obtidas com o uso da ArduEasy, onde classificaram majoritariamente como “fácil” ou “muito fácil”. Esse resultado sugere que a abstração fornecida pela ArduEasy teve um impacto positivo significativo na resolução do problema, ao reduzir a complexidade percebida pelos participantes.

Já quando questionados sobre a dificuldade na organização e escrita do código, boa parte das respostas se concentraram em “difícil” e “média” (44,4% cada) e apenas 11,11% como “fácil”, novamente evidenciando um contraste com a biblioteca ArduEasy, onde a maioria dos participantes (55,5%) classificou como “fácil” ou “muito fácil”. Esses dados reforçam a ideia de que a ArduEasy teve um papel importante na redução da carga cognitiva associada à estruturação do código. A organização e escrita do código, que muitas vezes envolve compreender a lógica de controle, nomeação adequada de variáveis e funções, e sequenciamento das ações, é um dos maiores desafios para iniciantes em programação. O fato de que a maioria dos par-

participantes considerou esse quesito como médio ou difícil com a API padrão, mas fácil com a ArduEasy, sugere que a biblioteca conseguiu oferecer uma interface mais intuitiva, provavelmente por meio de comandos mais claros, menos código repetitivo e abstração de detalhes de mais baixo nível.

Quanto à facilidade em aprender e aplicar os conceitos da API padrão, 33,3% classificou como fácil, 55,6% como médio e apenas 11,1% como difícil. Isso sugere que, apesar das dificuldades, os principais conceitos da API padrão, como as funções básicas abordadas (*pinMode*, *digitalWrite*, *digitalRead*, *delay*, *millis*, etc.), foram compreendidos pelos participantes. Porém, fica evidente que, embora tenham compreendido, os resultados dos parágrafos anteriores indicam uma maior dificuldade de pôr em prática.

Quando questionados sobre a facilidade em gerenciar o tempo utilizando a API padrão, todas as respostas se concentraram em “médio” (44,4%) e “difícil” (55,6%), indicando que este aspecto foi o mais desafiador. Isso sugere que a técnica apresentada para o controle de tempo não bloqueante (uso da função *millis*, variáveis de controle e comparação temporal) não é intuitiva para iniciantes, podendo-se tornar verbosa e até confusa, especialmente em projetos que envolvem múltiplos trechos de esperas não bloqueantes. Por outro lado, usando a biblioteca ArduEasy, todos os participantes classificaram como fácil ou muito fácil essa mesma tarefa de gerenciamento e controle de tempo.

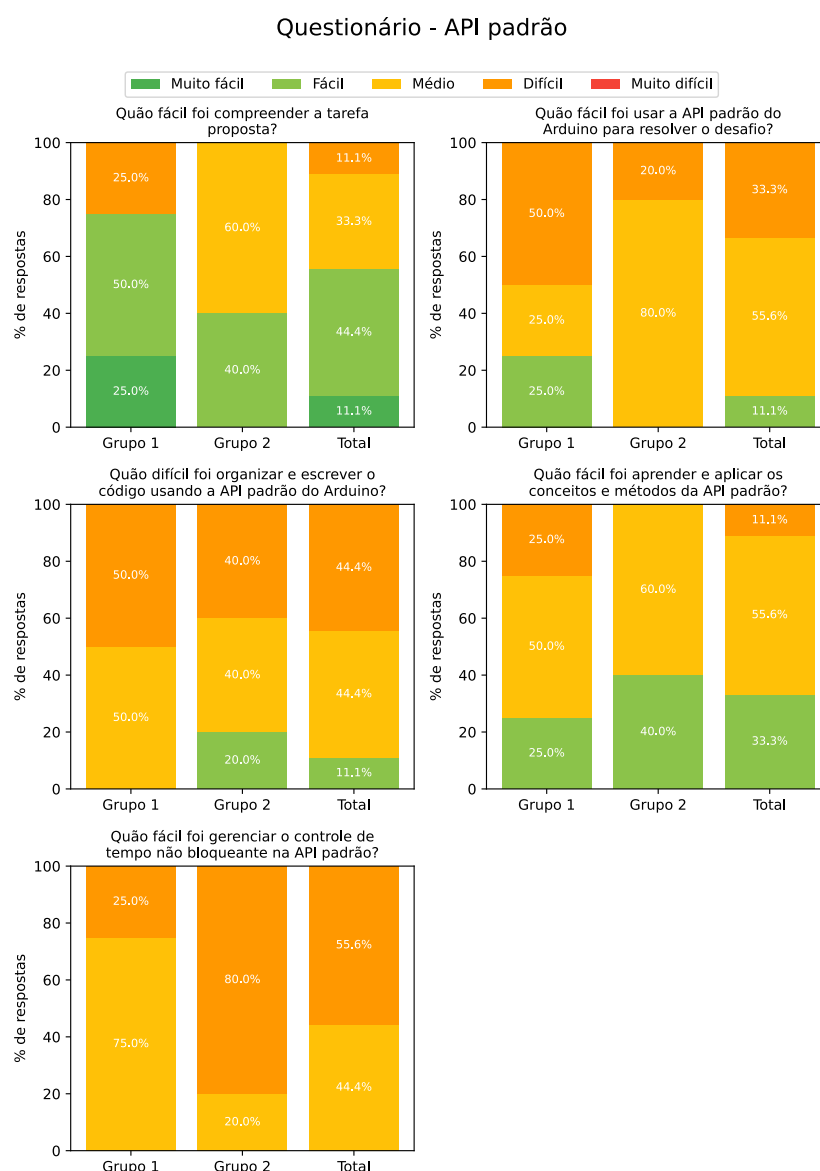


Figura 5.2: Gráficos das respostas do formulário de avaliação da API padrão. Fonte: De autoria própria.

As respostas qualitativas (Quadros 5.4 e 5.5) trazem detalhes importantes. Quando perguntados sobre o maior desafio enfrentado (Quadro 5.4), os participantes citaram com frequência o “controle de tempo” (66%) e a dificuldade de “formular a lógica” (33%). Essa percepção também aparece no Quadro 5.5, onde os participantes explicam quais partes foram particularmente difíceis, citando também o “controle de tempo” (62,5%) e “formulação da lógica” (25%). Esses relatos reforçam os apontamentos feitos nos parágrafos anteriores, evidenciando a complexidade percebida nesses aspectos ao utilizar a API padrão.

Qual foi o maior desafio que você enfrentou ao usar a API padrão do Arduino?	Grupo
Controlar o tempo.	Grupo 2
Controle de tempo	Grupo 2
Formular a lógica	Grupo 1
formular a logica	Grupo 1
Determinar a função de cada elemento	Grupo 1
Determinar o controle do tempo.	Grupo 1
controle de tempo	Grupo 2
Controle de tempo	Grupo 2
Controle de tempo	Grupo 2

Quadro 5.4: Respostas de “Qual foi o maior desafio que você enfrentou ao usar a API padrão do Arduino?”. Fonte: De autoria própria.

5.2.1 Alguma parte da tarefa foi particularmente difícil? Por quê?

Alguma parte da tarefa foi particularmente difícil? Por quê?	Grupo
Não.	Grupo 2
Na hora de programar o código	Grupo 2
Sim, ajustar o tempo	Grupo 1
sim, ajustar o tempo	Grupo 1
determinar a função de cada elemento	Grupo 1
Sim, a parte de determinar o tempo de cada coisa.	Grupo 1
Controle de tempo	Grupo 2
a gestão do controle de tempo	Grupo 2

Quadro 5.5: Respostas de “Alguma parte da tarefa foi particularmente difícil? Por quê?”. Fonte: De autoria própria.

Esses resultados mostram, de maneira geral, uma maior dificuldade na resolução do desafio utilizando apenas a API padrão do Arduino em comparação à implementação com a biblioteca ArduEasy, especialmente em relação ao controle de tempo de forma não bloqueante.

5.3 Avaliação comparativa entre as abordagens

O questionário comparativo revelou uma preferência evidente pela biblioteca ArduEasy. Na Figura 5.3, observa-se que:

- A ArduEasy foi considerada mais fácil de entender e utilizar.
- Produziu um código mais simples e legível.
- Permitiu resolver o desafio com maior rapidez.
- E foi a abordagem preferida por todos os participantes para futuros projetos.

Questionário - Comparativo

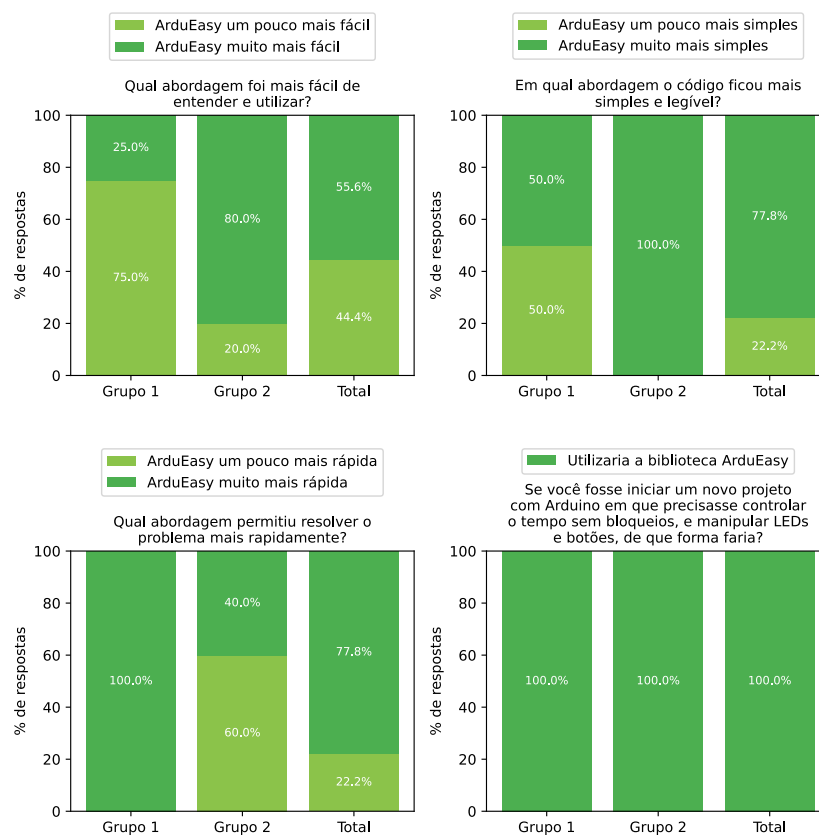


Figura 5.3: Gráficos das respostas do formulário de avaliação comparativa. Fonte: De autoria própria.

Embora os participantes tenham sido divididos em grupos com ordens diferentes de exposição (ArduEasy primeiro e API padrão depois, e vice-versa), os dados indicam que a biblioteca ArduEasy proporcionou benefícios percebidos de forma consistente. Mesmo os participantes que utilizaram primeiro a ArduEasy destacaram suas vantagens com clareza. Isso sugere que os ganhos oferecidos pela biblioteca são robustos e não estão condicionados à ordem de apresentação dos conteúdos.

Esses dados corroboram a hipótese central deste trabalho, demonstrando que a abstração proposta contribui significativamente para a experiência de iniciantes com Arduino.

6

Conclusão

Este trabalho apresentou o desenvolvimento e a avaliação da biblioteca ArduEasy, uma proposta de abstração voltada a facilitar o uso da plataforma Arduino em aplicações diversas. Seu principal objetivo é oferecer uma camada intermediária de controle que permita aos usuários realizarem tarefas comuns de maneira mais simples e intuitiva.

Para validar a proposta, foram realizados experimentos com dois grupos de participantes sem experiência anterior com Arduino. Os resultados indicaram que a biblioteca foi eficaz em reduzir a complexidade de implementação do desafio proposto.

Embora o experimento tenha sido conduzido com iniciantes, a ArduEasy foi desenvolvida com foco em uso geral, podendo beneficiar desde usuários em fase de aprendizado até desenvolvedores que desejam acelerar a prototipagem ou criar aplicações mais legíveis e organizadas. A estrutura modular da biblioteca, aliada a uma interface unificada para diferentes componentes, contribui para tornar o código mais limpo e menos propenso a erros comuns, como manipulação incorreta de pinos. Além disso, as interfaces introduzidas pela biblioteca permitem que usuários mais experientes estendam as funcionalidades da ArduEasy por meio de suas próprias implementações, favorecendo a contribuição da comunidade para a evolução contínua da biblioteca.

Apesar dos resultados positivos, algumas limitações técnicas foram identificadas e apontam direções claras para melhorias futuras. A principal diz respeito ao uso atual de listas encadeadas com alocação dinâmica de memória, uma escolha que, embora flexível, pode levar à fragmentação de memória em plataformas com recursos mais restritos, como os microcontroladores baseados na arquitetura AVR. Como continuidade deste trabalho, propõe-se a substituição dessa estrutura por alternativas estáticas ou híbridas, que mantenham a flexibilidade sem comprometer a estabilidade em sistemas embarcados de baixa capacidade.

Outra frente importante para a evolução da ArduEasy consiste na coleta de feedbacks de usuários mais experientes na plataforma Arduino. A escuta ativa desse público pode oferecer *insights* valiosos sobre funcionalidades desejadas, dificuldades não percebidas durante os testes com iniciantes, além de sugestões de novos dispositivos que poderiam ser contemplados

por meio de classes específicas de controle. Essa aproximação com a comunidade usuária permitirá guiar a evolução da biblioteca com base em demandas reais, tornando-a mais robusta, abrangente e adaptada a diferentes contextos de uso.

Em síntese, a ArduEasy representa uma contribuição promissora para o ecossistema Arduino ao buscar equilíbrio entre simplicidade de uso e flexibilidade. Com os aprimoramentos futuros propostos, espera-se que a biblioteca se consolide como uma ferramenta útil e confiável para a comunidade Arduino em suas múltiplas frentes de aplicação.

Por fim, em conformidade com os objetivos estabelecidos no início deste trabalho, a biblioteca ArduEasy foi disponibilizada publicamente sob a licença GPLv3 em um repositório aberto (<https://github.com/jmksn/ArduEasy>) e publicada no gerenciador oficial de bibliotecas do projeto Arduino (<https://docs.arduino.cc/libraries/ardueasy/>), permitindo seu uso, estudo, modificação e redistribuição por qualquer interessado.

Referências

ADAFRUIT LEARNING SYSTEM. **Adafruit Motor Shield V2**. 2013. Disponível em: <<https://learn.adafruit.com/adafruit-motor-shield-v2-for-arduino/overview>>. Acesso em: 26 ago. 2024.

_____. **LEDs**. 2012. Disponível em: <<https://learn.adafruit.com/adafruit-arduino-lesson-2-leds/leds>>. Acesso em: 1 ago. 2024.

ARDUINO DOCUMENTATION. **ATmega168/328P-Arduino Pin Mapping**. Disponível em: <<https://docs.arduino.cc/retired/hacking/hardware/PinMapping168/>>. Acesso em: 5 ago. 2024.

_____. **Bootloader**. 2024. Disponível em: <<https://docs.arduino.cc/retired/hacking/software/Bootloader/>>. Acesso em: 11 mai. 2024.

_____. **UNO R3**. Disponível em: <<https://docs.arduino.cc/hardware/uno-rev3/#features>>. Acesso em: 3 ago. 2024.

_____. **What is Arduino?** 2022. Disponível em: <<https://docs.arduino.cc/learn/starting-guide/whats-arduino/>>. Acesso em: 11 mai. 2024.

ARDUINO HARDWARE. **Arduino Hardware**. Disponível em: <<https://www.arduino.cc/en/hardware>>. Acesso em: 11 mai. 2024.

BANZI, Massimo; SHILOH, Michael. **Make: Getting Started with Arduino**. 3. ed. [S.l.]: Maker Media, 2014. ISBN 9781449363338.

CADY, F.M. **Microcontrollers and Microcomputers: Principles of Software and Hardware Engineering**. 2. ed. [S.l.]: Oxford University Press, 2009. ISBN 9780195371611.

DUNBAR, Norman. **Arduino Software Internals: A complete guide to how your Arduino language and hardware work together**. [S.l.]: Apress, 2020. ISBN 9781484257906.

ELETRONICWINGS. **DC Motor Basics, Working Principle, interfacing with Arduino**. Disponível em: <<https://www.electronicwings.com/sensors-modules/dc-motor>>. Acesso em: 1 ago. 2024.

GAMMA, Erich et al. **Design Patterns: Elements of Reusable Object-Oriented Software**. [S.l.]: Addison-Wesley Professional, 1994. ISBN 9780201633610.

GRIDLING, Günther; WEISS, Bettina. **Introduction to Microcontrollers**. [S.l.]: Vienna University of Technology, 2007.

GUPTA, Sourav. **Pull Up and Pull Down Resistor**. 2018. Disponível em: <<https://circuitdigest.com/tutorial/pull-up-and-pull-down-resistor>>. Acesso em: 3 ago. 2024.

LOBO DA ROBÓTICA. **Como usar o DHT11 com Arduino?** Disponível em: <<https://lobodarobotica.com/blog/ler-temperatura-e-umidade-com-arduino-e-o-dht11/>>. Acesso em: 1 ago. 2024.

MICROCHIP. **ATMEGA328P**. Disponível em: <<https://www.microchip.com/en-us/product/atmega328p>>. Acesso em: 3 ago. 2024.

_____. **AVR® MCU Digital I/O Ports**. 2023. Disponível em: <<https://developerhelp.microchip.com/xwiki/bin/view/products/mcu-mpu/8-bit-avr/structure/ioports/>>. Acesso em: 4 ago. 2024.

MICROSHIP. **megaAVR® Data Sheet**. [S.l.], 2018.

MOTA, Allan. **O que é Arduino e como funciona?** 2021. Disponível em: <<https://portal.vidadesilicio.com.br/o-que-e-arduino-e-como-funciona/>>. Acesso em: 3 ago. 2024.

PASQUALE, Alyssa J. **Microcontrollers**. [S.l.]: College of DuPage, 2024.

PICORETI, Rodolfo. **Teclado Matricial e Multiplexação**. 2017. Disponível em: <<https://portal.vidadesilicio.com.br/teclado-matricial-e-multiplexacao/>>. Acesso em: 15 dez. 2024.

RANELLUCCI, Alessandro. **Arduino Language**. 2022. Disponível em: <<https://github.com/arduino/language>>. Acesso em: 6 ago. 2024.

SARAFAN, Randy. **Modules and Shields**. Disponível em: <<https://www.instructables.com/Modules-and-Shields/>>. Acesso em: 3 ago. 2024.

SCHNEIDER, Josh. **Microcontrollers vs. microprocessors: What's the difference?** 2024. Disponível em: <<https://www.ibm.com/blog/microcontroller-vs-microprocessor/>>. Acesso em: 26 ago. 2024.

SILVA, Marcio. **Debounce, o que é e como resolver via software**. 2019. Disponível em: <https://www.makerhero.com/blog/debounce-o-que-e-e-como-resolver-via-software/?srsltid=AfmBOooizcCTjZQ99vxWVxvrhb2aPKkOfe6kPJKKL5oj_GKwh2JRPBiD>. Acesso em: 15 dez. 2024.

SÖDERBY, Karl. **Getting Started with Arduino**. 2024. Disponível em:

<<https://docs.arduino.cc/learn/starting-guide/getting-started-arduino/>>.

Acesso em: 11 mai. 2024.

SÖDERBY, Karl; HYLÉN, Jacob. **Getting Started with Arduino IDE 2**. 2024. Disponível

em: <<https://docs.arduino.cc/software/ide-v2/tutorials/getting-started-ide-v2/>>. Acesso em: 11 mai. 2024.

TECHZERO. **LDR Sensor**. Disponível em:

<<https://techzeero.com/sensors-modules/ldr-sensor/>>. Acesso em: 1 ago. 2024.

VALVANO, Jonathan W. **Embedded Systems: Introduction to ARM Cortex M microcontrollers**. 5. ed. [S.l.: s.n.], 2014. ISBN 9781477508992.

VOGEL, Peter. **Best Practices for Designing a Fluent API**. 2014. Disponível em:

<<https://visualstudiomagazine.com/articles/2013/12/01/best-practices-for-designing-a-fluent-api.aspx>>. Acesso em: 7 ago. 2024.

WHEAT, Dale. **Arduino Internals**. [S.l.]: Apress, 2011. ISBN 9781430238829.

Apêndice A

Slides do minicurso

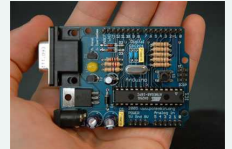
Os slides utilizados durante a apresentação do minicurso estão incluídos nas páginas a seguir.

A.1 Introdução ao Arduino



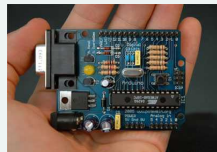
O QUE É ARDUINO?

- Uma plataforma para a criação de protótipos de circuitos eletrônicos
- Foi criado em 2005 por 5 pesquisadores na Itália
- O objetivo era criar um dispositivo barato, funcional e fácil de programar
- Atualmente é a principal plataforma para a criação de protótipos

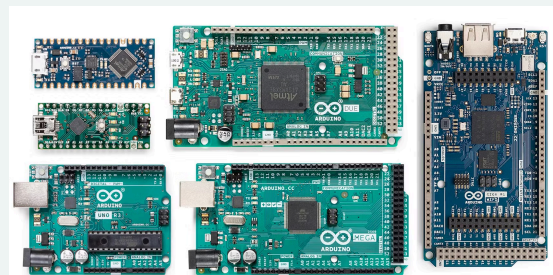


O QUE É ARDUINO?

- Suas placas são compostas por um **microcontrolador**, circuitos de entrada e saída e circuitos para a comunicação
- Pode ser programado através de uma IDE (Ambiente de Desenvolvimento Integrado) própria através da **linguagem de programação C/C++**
- Atualmente existem diversos modelos de placas para diferentes propósitos, sendo o **Arduino UNO** a versão mais popular

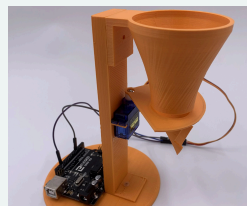


O QUE É ARDUINO?



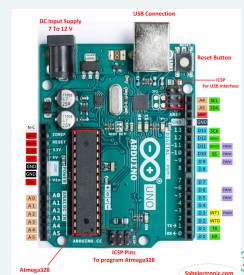
O QUE POSSO FAZER COM ARDUINO?

- Automatizar sua casa
 - Irrigação de jardins
 - Alimentador de pets
 - Controle de tomadas, lâmpadas, ar-condicionado, etc...
- Criar um novo brinquedo ou equipamento
- As possibilidades são infinitas



COMO O ARDUINO INTERAGE COM O MUNDO EXTERNO?

- As placas Arduino possuem pinos que permitem a conexão de dispositivos de entrada e saída
- Dispositivos de entrada emitem sinais elétricos para os pinos
- Dispositivos de saída recebem sinais elétricos dos pinos



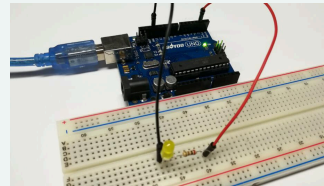
COMO O ARDUINO INTERAGE COM O MUNDO EXTERNO?

- **Sensores:** Dispositivos de entrada que detectam alguma interação vinda do mundo externo e converte em um sinal elétrico, como o pressionamento de um botão
- **Atuadores:** Dispositivos de saída que transformam um sinal elétrico em uma ação física, como um LED por exemplo.



COMO O ARDUINO INTERAGE COM O MUNDO EXTERNO?

- Alguns dispositivos necessitam de um circuito extra para ser ligado ao Arduino

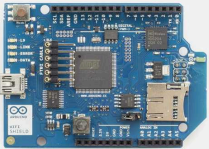


- Pode-se usar **protoboard** para facilitar a montagem de circuitos sem a necessidade de soldagem



COMO O ARDUINO INTERAGE COM O MUNDO EXTERNO?

- Existem também circuitos já prontos para serem ligados ao Arduino, como **shields** e módulos



COMO PROGRAMAR O ARDUINO?

- Através do programa de computador chamado **Arduino IDE** (Ambiente de Desenvolvimento Integrado)
- Utilizando as **linguagens de programação C/C++**



O QUE É UM PROGRAMA?

- Um conjunto de instruções que descrevem uma tarefa a ser entendida e realizada por um dispositivo computacional
- São escritos através de uma **linguagem de programação**, como C, C++, Java e Python
- A grosso modo, são os arquivos **.exe** do Windows por exemplo



O QUE É UMA LINGUAGEM DE PROGRAMAÇÃO?

- Um conjunto de símbolos, palavras e regras que são utilizadas para a criação de um algoritmo/programa.
- São mais próximas da linguagem natural (a de nós humanos :)
- O programa é escrito através de um código-fonte (arquivo de texto)
- O código-fonte de uma linguagem precisa ser "traduzido" para que o computador (Arduino, celulares, etc...) entenda



O QUE É UMA LINGUAGEM DE PROGRAMAÇÃO?

- Essa "tradução" pode ser "ao vivo", ou seja, o código-fonte vai sendo convertido em linguagem de máquina a medida em que vai sendo executado
- A "tradução" pode ser feita antes da execução através de um programa chamado **compilador**



INTRODUÇÃO À LINGUAGEM C++



C++

- Foi criada no início dos anos 80
 - Inicialmente era uma extensão da linguagem C
 - Projetada para ser compatível com a linguagem C
- É uma linguagem de alto nível, ou seja, está mais próxima da linguagem humana do que da linguagem de máquina
- É uma linguagem compilada
- Cada sentença (comando a ser executado) deve ser seguido por um ";"



VARIÁVEIS

- Representam um pedacinho da memória do computador e podemos utilizá-las para salvar alguma informação a ser usada pelo programa
- São compostas por um identificador (nome) e um tipo, que define o que essa variável pode armazenar



VARIÁVEIS

- Principais tipos:
 - char: Armazena um único caractere ('A')
 - int: Armazena um número inteiro
 - long: Armazena números inteiros muito grande
 - float: Armazena números reais
 - double: Armazena números reais bem maiores ou com melhor precisão
 - string: Armazenam um texto
 - bool: Armazenam um valor lógico ("true" ou "false")
- O programador também pode criar seus próprios tipos



VARIÁVEIS

- Exemplos:

```
int idade = 5; // (int é o tipo e "idade" é o nome da variável)
string nome = "Maria";
char genero = 'F';
```

```
// Variáveis declaradas com "const" possuem um valor constante e não
// podem ser alteradas.
const float pi = 3.14159;
```

```
int dobro_da_idade; // Podemos criar variável sem definir um valor
dobro_da_idade = idade * 2;
```



EXPRESSÕES LÓGICAS E ARITIMÉTICAS

- A linguagem permite a avaliações de expressões lógicas e aritméticas
- Expressões aritméticas utilizam os mesmos operadores da matemática
 - + - * /
- Operadores relacionais
 - == : Igualdade (1 == 5 | a == 10 | 3 == b)
 - != : Diferente (1 != 5 | a != 10 | genero != 'F')
 - >, >=, <, <= : Semelhantes ao da matemática (1 >= 0 | a < 10)



EXPRESSÕES LÓGICAS E ARITIMÉTICAS

- Operadores lógicos

Operador	Significado	Exemplo
&&	"E" lógico, verdadeiro quando ambas expressões são verdadeiras	bool debutante = idade == 15 && genero == 'F';
	"Ou" lógico, verdadeiro quando ao menos uma das expressões for verdadeira	bool reprovado = nota < 7 porcentagem_presenca < 75
!	"Não" lógico, verdadeiro quando o operando for falso	bool nao_pode_dirigir = !tem_cnh;



ESTRUTURAS DE DECISÃO

- Permite decidirmos se uma sequência de comandos será executada com base em uma condição
- Uma das formas de fazer isso em C++ é através do comando **if**
 - Pode ser usado o também comando **else** para quando a condição do **if** for falsa
- Há também o comando **switch**, mas não será abordado aqui.

```
if (debutante) {  
    cout << "Parabéns debutante!";  
} else {  
    cout << "Parabéns";  
}
```



ESTRUTURAS DE REPETIÇÃO

- Permite a repetição de uma sequência de comandos
- Pode ser feito através dos comandos:
 - **for**:
 - for (<inicialização>; <condição>; <incremento>) {
 - // comandos
 - }
 - **while**:
 - while (<condição>) {
 - // comandos
 - }



ESTRUTURAS DE REPETIÇÃO

- Permite a repetição de uma sequência de comandos
- Pode ser feito através dos comandos:
 - **do-while**:
 - do {
 - // comandos
 - } while(<condição>);



ESTRUTURAS DE REPETIÇÃO

- Exemplo:
- Como Exibir 10 vezes o texto "Olá mundo"?

```
for (int i = 0; i < 10; i++) {  
    cout << "Olá mundo";  
}
```

```
int i= 0;  
while (i < 10) {  
    i++;  
    cout << "Olá mundo";  
}
```

```
int i= 0;  
do {  
    i++;  
    cout << "Olá mundo";  
} while (i < 10);
```

Obs.: i++; é o mesmo que: i = i + 1;



FUNÇÕES

- Funções são blocos de código que podem:
 - Executar outras funções
 - Receber ou não valores através de variáveis conhecidas como **parâmetros**
 - Retornar ou não um valor
 - Quando não retornam nenhum valor, seu tipo de retorno deverá ser **void**
 - Permitem o reuso de código, já que as funções podem ser executadas em diferentes lugares
 - Um programa C++ deve conter ao menos uma função chamada **main**
- É criada ou declarada da seguinte forma:

```
<tipo de retorno> <nome da funcao>(<parâmetros, se houverem>) {  
    // código a ser executado  
    return <valor a ser retornado>; // Essa linha pode ser omitida caso não  
    // retorne nada  
}
```



FUNÇÕES

- Exemplos

```
void exibir() {  
    cout << "Olá mundo!";  
}  
  
int main() {  
    for (int i = 0; i < 10; i++) {  
        exibir();  
    }  
    return 0;  
}
```

```
float dobro (float x) {  
    return x * 2;  
}  
  
int main() {  
    float numero, dobro_numero;  
  
    cout << "Digite um número: ";  
    cin >> numero;  
  
    dobro_numero = dobro(numero);  
  
    cout << "O dobro de " << numero << " é: " << dobro_numero;  
  
    return 0;  
}
```



FUNÇÕES

- Para programar o Arduino precisamos obrigatoriamente criar duas funções:
- **setup**:
 - Função que é executada apenas uma vez, assim que a placa é ligada.
 - Ideal para fazer a configuração inicial dos dispositivos, quando necessário
- **loop**:
 - Função que é executada infinitamente.
 - É nela que "damos vida" ao Arduino e que adicionamos toda a lógica ou chamamos as demais funções que são responsáveis pela execução da tarefa desejada



FUNÇÕES

- Não precisamos criar a função **main** no Arduino, ela já é implementada pela biblioteca do Arduino.

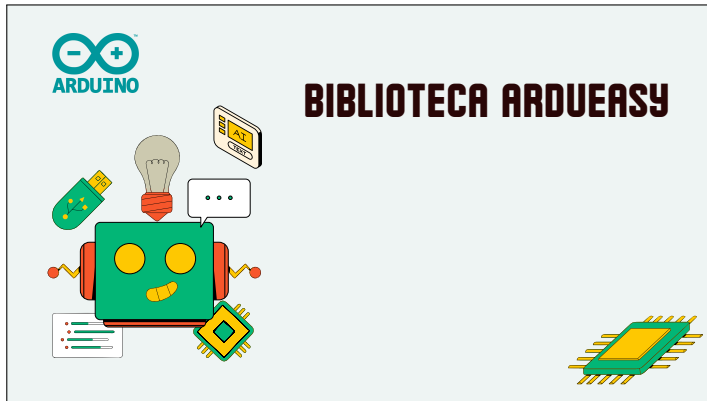


BIBLIOTECAS

- É um conjunto de funções, tipos e constantes externas que podemos adicionar ao nosso código
- Também chamadas de **lib** (library) ou **API** (interface de programação de aplicações)
- É útil para não termos que reinventar a roda
 - Podemos usar um código já existente criado por você ou terceiros
- Em C++ importamos uma biblioteca através do comando (macro) **#include "<nome da biblioteca>"**
- Abordaremos duas bibliotecas aqui:
 - Biblioteca padrão do Arduino
 - ArduEasy



A.2 Biblioteca ArduEasy



ARDUEASY

- Auxilia a manipulação de alguns dispositivos e o controle de tempo
- Foi criada com base na biblioteca padrão do Arduino
- Permite a manipulação de:
 - Leds
 - Botões
 - Teclados matriciais
 - Entrada analógica
 - Motores DC (corrente contínua)



COMO UTILIZÁ-LA?

- Primeiro devemos importar a biblioteca através do comando **#include "ArduEasy.h"**
- Em seguida, devemos chamar a função **ArduEasy::run();** dentro da função **loop**, conforme o exemplo a seguir:

```
#include "ArduEasy.h"

void setup() {
}

void loop() {
  ArduEasy::run();
}
```



COMO UTILIZÁ-LA?

- Utilize a função **setup** como ponto de partida para configurar os dispositivos ou componentes da biblioteca



O TIPO LED

- A biblioteca possui um tipo de variável chamada **Led**, com ele é possível ligar, desligar e piscar LEDs
- Podemos criar uma variável do tipo **Led** da seguinte forma:
 - **Led led(<pino ao qual o led está ligado>);**
- As funções do tipo **Led** são:
 - **turnOn();** Liga um led
 - **turnOff();** Desliga um led
 - **toggle();** Inverte o estado do led
 - Liga quando desligado
 - Desliga quando ligado
 - **blink(<tempo ligado>, <tempo desligado>);** Pisca um led
 - Poderá ser especificado o tempo em ms em que o led estará ligado e desligado, sendo 500ms o valor padrão.



EXEMPLO 1: COMO ACIONAR UM LED

```
#include "ArduEasy.h"

Led led(13);

void setup() {
  led.turnOn();
}

void loop() {
  ArduEasy::run();
}
```



EXEMPLO 2: COMO PISCAR UM LED

```
#include "ArduEasy.h"

Led led(13);

void setup() {
  led.blink(300, 200);
}

void loop() {
  ArduEasy::run();
}
```



O TIPO BUTTON

- O tipo **Button** permite definirmos uma ação a ser executada quando um botão for pressionado
- Podemos criar uma variável do tipo Button da seguinte forma:
 - Button button(<pino ao qual o botão está ligado>);
- É possível definir a ação a ser executada usando a função **onPress(<acao>)**
 - A ação poderá ser outra função, exemplo:
 - Supondo que você tenha criado a função **ao_pressionar() { ... }**
 - **botao.onPress(ao_pressionar);**
 - Obs.: Usar apenas o nome da função no parâmetro (sem os parênteses)
 - A ação pode ser definida como uma função anônima:
 - **botao.onPress([]{ ... });**



EXEMPLO: INVERTER LED AO PRESSIONAR BOTÃO

```
#include "ArduEasy.h"

Led led(13);
Button botao(2);

void setup() {
  botao.onPress([]{
    led.toggle();
  });
}

void loop() {
  ArduEasy::run();
}
```



O TIPO ACTIONS

- O tipo **Actions** permite definirmos uma sequência de ações com um intervalo de tempo (ou não) entre elas
- Podemos criar uma variável do tipo Actions da seguinte forma:
 - Actions acoes(<bool indicando se as ações se repetirão após o término>);
- As principais funções do tipo Actions são:
 - **execute()**: Faz com que as ações configuradas sejam executadas
 - **enable()**: Habilita a execução das ações
 - **disable()**: Desabilita a execução das ações
 - **doAndWait**(<funcao a ser executada>, <tempo de espera>): Executa uma função e espera o tempo informado para então executar a próxima ação
 - **stop()**: Para a execução das ações e define a primeira ação configurada como próxima ação
 - **isRunning()**: Retorna um bool indicando se as ações estão sendo executadas



ALTERNAR DOIS LEDS EM TEMPOS DIFERENTES

```
#include "ArduEasy.h"

Led led1(9);
Led led2(10);

Actions actions(true);

void liga_led1_apenas() {
  led1.turnOn();
  led2.turnOff();
}

void liga_led2_apenas() {
  led2.turnOn();
  led1.turnOff();
}

void setup() {
  actions.doAndWait(liga_led1_apenas, 5000)
  .doAndWait(liga_led2_apenas, 3000)
  .execute();
}

void loop() {
  ArduEasy::run();
}
```



COMO ATIVAR/DESATIVAR A ALTERNÂNCIA DE DOIS LEDS COM UM BOTÃO

```
#include "ArduEasy.h"

Led led1(9);
Led led2(10);
Button botao(2);

Actions actions(true);

void liga_led1_apenas() {
  led1.turnOn();
  led2.turnOff();
}

void liga_led2_apenas() {
  led2.turnOn();
  led1.turnOff();
}

void setup() {
  actions.doAndWait(liga_led1_apenas, 5000)
  .doAndWait(liga_led2_apenas, 3000)
  .execute();

  botao.onPress([]{
    if (actions.isRunning()) {
      actions.stop();
      led1.turnOff();
      led2.turnOff();
    } else {
      actions.execute();
    }
  });

  void loop() {
    ArduEasy::run();
  }
}
```



DESAFIO:

- Programe o Arduino para controlar semáforos
- Seu semáforo poderá ser ativado ou desativado quando um botão for pressionado
- Quando **ativo** utilize os seguintes tempos para a transição de cores:
 - **Verde:** ficará ligado por **5 segundos**
 - **Amarelo:** ficará ligado por **3 segundos**
 - **Vermelho:** ficará ligado por **8 segundos**
- Quando **desativado**, todos os leds deverão permanecer apagados



A.3 Biblioteca padrão do Arduino



BIBLIOTECA PADRÃO

- Faz parte do projeto Arduino e já vem embutida na IDE
- Possui funções já implementadas, como:
 - Manipulação de pinos
 - Definir um pino como entrada ou saída
 - Emitir e ler um sinal digital em um pino
 - Emitir e ler sinal analógico em um pino
 - Operações matemáticas
 - Calcular valor absoluto
 - Potência
 - Raiz quadrada
 - Seno, cosseno e tangente



BIBLIOTECA PADRÃO

- Faz parte do projeto Arduino e já vem embutida na IDE
- Possui funções já implementadas, como:
 - Controle de tempo
 - Pausar o programa por um determinado tempo
 - Obter a quantidade de tempo em que o programa está sendo executado



COMO LER E ENVIAR SINAIS DIGITAIS NOS PINOS?

- A função **pinMode(<pino>, <modo>)** permite definirmos qual será o modo de operação de um pino
 - **INPUT:** O pino atuará como uma entrada
 - **OUTPUT:** O pino atuará como uma saída
- A função **digitalWrite(<pino>, <estado>)** permite enviarmos um sinal digital para um pino de saída
 - **HIGH:** Configura uma tensão alta no pino (liga)
 - **LOW:** Configura uma tensão baixa no pino (desliga)
- A função **digitalRead(<pino>)** verifica e retorna o estado atual do pino, que será:
 - **HIGH:** O pino está detectando uma tensão alta
 - **LOW:** O pino não está detectando uma tensão baixa



EXEMPLO 1: COMO ACIONAR UM LED

```
const int led = 13;

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

void loop() {
  digitalWrite(led, HIGH);
}
```



EXEMPLO 2: LIGAR UM LED QUANDO UM BOTÃO ESTIVER PRESSIONADO

```
const int led = 13;
const int botao = 2;

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

void loop() {
  int estadoBotao = digitalRead(botao);
  digitalWrite(led, estadoBotao);
}
```



COMO FAZER UM LED PISCAR?

- Precisamos manter o LED ligado por um tempo e também desligá-lo por um período
- A função **delay(<tempo em ms>)** permite pausar a execução do programa

```
const int led = 13;

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

void loop() {
  digitalWrite(led, HIGH);
  delay(300);
  digitalWrite(led, LOW);
  delay(200);
}
```



COMO (NÃO) PISCAR DOIS LEDS EM TEMPOS DIFERENTES?

```
const int led1 = 13;
const int led2 = 10;
```

```
void setup() {
  pinMode(led1, OUTPUT);
  pinMode(led2, OUTPUT);
}
```

```
void loop() {
  digitalWrite(led1, HIGH);
  delay(300);
  digitalWrite(led1, LOW);
  delay(300);
```

```
  digitalWrite(led2, HIGH);
  delay(200);
  digitalWrite(led2, LOW);
  delay(200);
}
```



CONTROLE DE TEMPO SEM BLOQUEIO

- A função **delay** é muito útil, porém trava todo o programa enquanto espera o tempo estipulado
- Podemos utilizar a função **millis()** para saber quanto tempo se passou desde um momento desejado
- Ex:

```
const int led = 13;

long tempo = millis();

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

void loop() {
  if ((millis() - tempo) > 5000) {
    digitalWrite(led, HIGH);
  }
}
```



COMO PISCAR DOIS LEDS EM TEMPOS DIFERENTES?

```
const int led1 = 13;
const int led2 = 10;
```

```
long tempo_led1 = 0;
long tempo_led2 = 0;
bool led1_ligado = false;
bool led2_ligado = false;
```

```
void setup() {
  pinMode(led1, OUTPUT);
  pinMode(led2, OUTPUT);
}
```

```
void loop() {
  if (millis() - tempo_led1 > 300) {
    led1_ligado = !led1_ligado;
    tempo_led1 = millis();
  }
```

```
  if (millis() - tempo_led2 > 200) {
    led2_ligado = !led2_ligado;
    tempo_led2 = millis();
  }
```

```
  digitalWrite(led1, led1_ligado);
  digitalWrite(led2, led2_ligado);
}
```



COMO ALTERNAR DOIS LEDS SEM BLOQUEIO?

- Supomos que o Led1 ficará aceso por 5 segundos, enquanto o Led2 ficará aceso por 3 segundos
- Podemos usar uma variável para armazenar quando o ciclo iniciou
 - Sempre que o "tempo atual" – "ultimo reinicio" > 8 segundos significa que o ciclo encerrou
 - Se "tempo atual" – "ultimo reinicio" for < 5 segundos significa que o Led1 deve se manter aceso e Led2 apagado
 - Se "tempo atual" – "ultimo reinicio" for > 5 segundos significa que o Led2 deve se manter aceso e Led1 apagado



COMO ALTERNAR DOIS LEDS SEM BLOQUEIO?

```
const int led1 = 9;
const int led2 = 10;
```

```
long tempoRecomeco = 0;
```

```
void setup() {
  pinMode(led1, OUTPUT);
  pinMode(led2, OUTPUT);
}
```

```
void loop() {
  long agora = millis();
  long diferencaTempo = agora - tempoRecomeco;
```

```
  if (diferencaTempo > 8000) {
    tempoRecomeco = agora;
  }
```

```
  if (diferencaTempo > 5000) {
    digitalWrite(led1, HIGH);
    digitalWrite(led2, LOW);
  } else {
    digitalWrite(led1, LOW);
    digitalWrite(led2, HIGH);
  }
}
```



COMO VERIFICAR SE UM BOTÃO FOI PRESSIONADO?

- Precisamos ler o sinal digital presente no pino do botão
 - Se for **HIGH** há corrente passando pelo pino, logo o botão foi pressionado
 - Consequentemente **LOW** indica que o botão não está pressionado
- Ao pressionar um botão, o contato entre suas partes condutoras oscilam devido à energia cinética, esse fenômeno é conhecido como efeito **bounce**
 - Ou seja, imediatamente após pressionar um botão, o pino oscilará entre **HIGH** e **LOW** por um curto intervalo de tempo
- Para uma verificação precisa, precisaremos também tratar esse efeito



EXEMPLO: INVERTER LED AO PRESSIONAR BOTÃO

```
const int botao = 2;
const int led = 10;

long ultimoDebounce = 0;
int ultimoEstadoBotao = LOW;

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

void loop() {
  int estadoBotao = digitalRead(botao);
  if ((millis() - ultimoDebounce > 50) && estadoBotao != ultimoEstadoBotao) {
    ultimoDebounce = millis();
    ultimoEstadoBotao = estadoBotao;

    if (estadoBotao == HIGH) {
      digitalWrite(led, digitalRead(led));
    }
  }
}
```



COMO ATIVAR/DESATIVAR A ALTERNÂNCIA DE DOIS LEDS COM UM BOTÃO

```
const int botao = 2;
const int led1 = 9;
const int led2 = 10;

long ultimoDebounce = 0;
int ultimoEstadoBotao = LOW;

long tempoRecomeco = 0;
bool ativo = true;

void setup() {
  pinMode(botao, INPUT);
  pinMode(led1, OUTPUT);
  pinMode(led2, OUTPUT);
}

void loop() {
  long agora = millis();
  long diferencaTempo = agora - tempoRecomeco;

  int estadoBotao = digitalRead(botao);
  if ((millis() - ultimoDebounce > 50) && estadoBotao != ultimoEstadoBotao) {
    ultimoDebounce = millis();
    ultimoEstadoBotao = estadoBotao;

    if (estadoBotao == HIGH) {
      ativo = !ativo;
      tempoRecomeco = agora;
    }

    if (ativo) {
      if (diferencaTempo > 8000) {
        digitalWrite(led1, HIGH);
        digitalWrite(led2, LOW);
      } else {
        digitalWrite(led1, LOW);
        digitalWrite(led2, HIGH);
      }
    } else {
      digitalWrite(led1, LOW);
      digitalWrite(led2, LOW);
    }
  }
}
```



DESAFIO:

- Programe o Arduino para controlar semáforos
- Seu semáforo poderá ser ativado ou desativado quando um botão for pressionado
- Quando **ativo** utilize os seguintes tempos para a transição de cores:
 - **Verde**: ficará ligado por **5 segundos**
 - **Amarelo**: ficará ligado por **3 segundos**
 - **Vermelho**: ficará ligado por **8 segundos**
- Quando **desativado**, todos os leds deverão permanecer apagados
- Dica:
 - É possível usar uma única variável para controlar o tempo de todos os leds
 - Faça em partes:
 - Primeiro faça os LEDs se alternarem com os tempos especificados
 - Use uma variável **bool** para indicar se está ativo ou não e teste usando valores fixos
 - Alterne os valores da variável do item acima sempre que o botão for pressionado.

