Fundação Universidade Federal de Mato Grosso do Sul
Faculdade de Computação - FACOM

SubProjeto DesTaCom
Despertando Novos Talentos em Computação no MS
Atividade Arduino

Equipe:
Hewerson Antonio Perdomo Jacquet
Luana Loubet Borges
Ricardo Espindola de Aguiar
Riccieli Kendy Zan Minakawa
Ricardo Ribeiro dos Santos

Sumário

1 Introdução **4**

2 O Kit de Desenvolvimento Arduino - Arduino MEGA 2560 **5**

2.1 Alimentação 6

2.2 Memória 7

2.3 Entrada e Saída 7

2.4 Comunicação 8

2.5 Reset Automático (Software) 8

2.6 Proteção contra sobrecorrente via USB 9

2.7 Características Físicas e Compatibilidade com *Shields* 9

3 Desenvolvimento de Programas para o Arduino **11**

3.1 Ambiente de Desenvolvimento 11

3.2 Programando para o Arduino: Conceitos e Sintaxe da Linguagem de Programação 14

3.2.1 Elementos de Sintaxe 14

3.2.2 Setup e Loop 16

3.2.3 Variáveis 17

3.2.4 Tipos de dados 17

3.2.5 Constantes 22

3.2.6 Conversão 24

3.2.7 Estrutura de Controle 26
3.2.8 Operadores de Comparação 30
3.2.9 Operador de Atribuição 30
3.2.10 Operadores Aritméticos 31
3.2.11 Operadores Booleanos 32
3.2.12 Operadores de Bits 32
3.2.13 Operadores Compostos 34
3.2.14 Entrada e saída digital 37
3.2.15 Entrada e saída analógica 38
3.2.16 Entrada e saída avançada 40
3.2.17 Tempo 42
3.2.18 Comunicação serial 44

4 Exemplo de Aplicação 1: Imprimindo uma mensagem no LCD 52

5 Exemplo de Aplicação 2: Alterando a frequência com que o LED pisca 54

6 Exemplo de Aplicação 3: Semáforo de Carros e Pedestres 56

7 Exemplo de Aplicação 4: Termômetro 59

8 Exemplo de Aplicação 5: Piano 63

9 Exemplo de Aplicação 6: Alarme 67

10 Exemplo de Aplicação 7: Projeto Alarme Multipropósito **69**

11 Exemplo de Aplicação 8: Portão Eletrônico **73**

1 Introdução

O projeto "Arduino"[1] teve início na cidade de Ivrea, Itália, em 2005, objetivando a efetivação de projetos de estudantes menos onerosa. Os fundadores do projeto, Massimo Banzi e David Cuartielles denominaram "Arduino" em homenagem a "Arduin de Ivrea" um antepassado histórico da cidade de Ivrea.

Arduino é um kit de desenvolvimento *open-source* baseado em uma placa de circuito impresso dotada de vários recursos de interfaceamento (pinagem de entrada e saída) e um microcontrolador Atmel AVR. É um projeto descendente da plataforma *Wiring* que foi concebida com o objetivo de tornar o uso de circuitos eletrônicos mais acessível em projetos multidisciplinares. A linguagem *Wiring* foi criada por Hernando Barragán em sua dissertação de mestrado no Instituto de Projetos Interativos de Ivrea sob a supervisão de Massimo Banzi e Casey Reas.

A linguagem usada para programação do Arduino é baseada na linguagem *Wiring* (sintaxe + bibliotecas), e muito similar a C++ com pequenas modificações. A linguagem adotada é baseada em *Processing*.

Atualmente, pode-se comprar um kit Arduino em diferentes versões. Também são disponibilizadas informações do hadware para aqueles que desejam montar seu próprio kit Arduino.

Além do ambiente de programação para o Arduino, existem outros softwares que podem facilitar o entendimento e documentação dessa tecnologia:

- Fritzing[2] é um ambiente de desenvolvimento de software dentro do projeto Arduino. Possibilita que os usuários possam documentar seus protótipos e, principalmente, que possam ilustrar a implementação de um projeto real de uma maneira fácil e intuitiva de ser entendida por outros usuários.

[1] http://arduino.cc
[2] http://fritzing.org/

- Miniblog[3] é um ambiente de desenvolvimento gráfico para Arduino. O principal objetivo é auxiliar o ensino de programação e, em especial, o ensino de robótica em nível de ensino médio.

2 O Kit de Desenvolvimento Arduino - Arduino MEGA 2560

O Arduino Mega 2560 é um kit de desenvolvimento baseado no microcontrolador ATmega2560 que possui 54 pino de entrada/saída (I/O), dos quais 14 podem ser usadas como saídas PWM de 8 bits, 16 entradas analógicas, 4 UARTs que são portas seriais de hardware, um cristal oscilador de 16MHz, uma conexão USB, um conector de alimentação, um conector ICSP (*In-Circuit Serial Programming*), e um botão de *reset*. Para energizar o Arduino Mega 2560 é necessário conectá-lo a um computador via cabo USB, a um adaptador AC/DC ou a uma bateria. Ressalta-se que a utilização do cabo USB é imprescindível quando deseja-se efetuar a programação do kit. O Arduino Mega 2560 (Figura 1) é uma evolução do Arduino Mega, que usa o microcontrolador ATmega 1280.

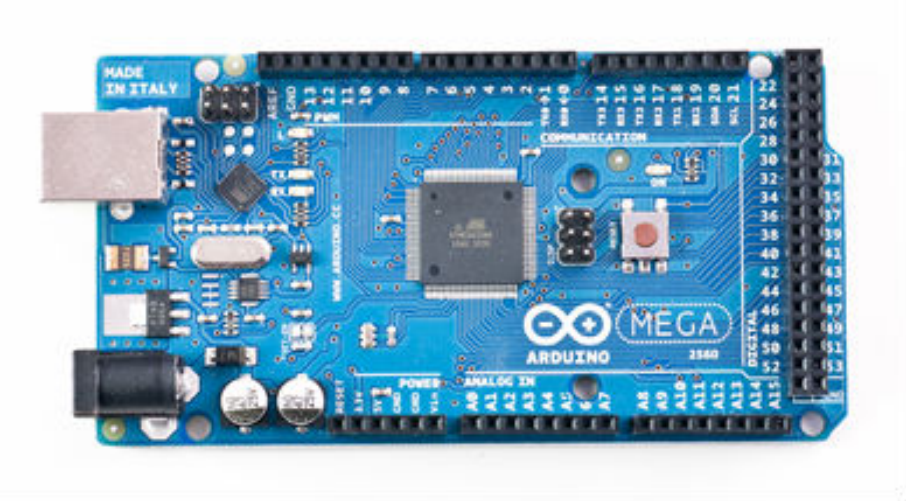

Figura 1: Arduino Mega 2560

A Tabela 1 resume todas as características já citadas e fornece algumas informações importantes a respeito da utilização do Arduino.

[3] http://blog.minibloq.org/

Microcontrolador	ATmega2560
Tensão de operação	5 V
Tensão de entrada (recomendada)	7-12 V
Tensão de entrada (limites)	6-20 V
Pinos de entrada e saída (I/O) digitais	54 (dos quais 14 podem ser saídas PWM)
Pinos de entradas analógicas	16
Corrente DC por pino I/O	40 mA
Corrente DC para pino de 3,3V	50 mA
Memória Flash	256 kB (dos quais 8 kB são usados para o *bootloader*)
SRAM	8 kB
EEPROM	4 kB
Velocidade de Clock	16 MHz

Tabela 1: Características do kit Arduino MEGA2560

2.1 Alimentação

O Arduino pode ser alimentado por uma conexão USB ou por uma fonte de alimentação externa que pode ser até uma bateria. A fonte pode ser ligada através de um conector de 2,1 mm (positivo no centro), na entrada de alimentação. Cabos vindos de uma bateria podem ser ligados nos pinos GND e entrada de alimentação (Vin) do conector de energia. A placa pode operar com alimentação externa entre 6 V e 20 V como especificado na Tabela 1. Entretanto, se a tensão aplicada for menor que 7 V, o regulador de 5 V pode fornecer menos de 5 V e a placa pode ficar instável. Com mais de 12 V o regulador de tensão pode superaquecer e danificar a placa. A faixa recomendável é de 7 V a 12 V.

Os pinos de alimentação são citados a seguir:

- **VIN**. Relacionado à entrada de tensão da placa Arduino quando se está usando alimentação externa (em oposição aos 5 volts fornecidos pela conexão USB ou outra fonte de alimentação regulada). É possível fornecer alimentação através deste pino ou acessá-la se estiver alimentando pelo conector de alimentação.
- **5V**. Fornecimento de tensão regulada para o microcontrolador e outros componentes da placa.
- **3V3**. Uma alimentação de 3,3 volts gerada pelo chip FTDI. A corrente máxima é de 50 mA.
- **GND**. Pinos de referência (0V).

2.2 Memória

O ATmega2560 tem 256 kB de memória flash para armazenamento de código (dos quais 8 kB é usado para o *bootloader*), 8 kB de SRAM e 4 kB de EEPROM (que pode ser lida e escrita com a biblioteca EEPROM).

2.3 Entrada e Saída

Cada um dos 54 pinos digitais do kit Arduino Mega 2560 pode ser usado como entrada ou saída, usando as funções de *pinMode()*, *digitalWrite()*, e *digitalRead()*. Eles operam a 5 volts. Cada pino pode fornecer ou receber uma corrente máxima de 40 mA e possui um resistor interno (desconectado por *default*) de 20-50kΩ.

Além disso, alguns pinos possuem funções especializadas:

- **Serial: 0 (RX) and 1 (TX); Serial 1: 19 (RX) and 18 (TX); Serial 2: 17 (RX) and 16 (TX); Serial 3: 15 (RX) and 14 (TX)**. Usados para receber (RX) e transmitir (TX) dados de forma serial com níveis TTL. Os Pinos 0 e 1 são conectados aos pinos correspondentes do chip ATmega8U2; que é um conversor USB-to-Serial.
- **Interruptores externos: 2 (interruptor 0), 3 (interruptor 1), 18 (interruptor 5), 19 (interruptor 4), 20 (interruptor 3), e 21 (interruptor 2)**. Estes pinos podem ser configurados para disparar uma interrupção por um nível lógico baixo, por uma transição descendente ou ascendente, ou por uma mudança de níveis Para mais detalhes deve-se ver a função *attachInterrupt()*.
- **PWM**: 0 a 13. Fornecem saída analógica PWM de 8 bits com a função *analogWrite()*.
- **SPI: 50 (MISO), 51 (MOSI), 52 (SCK), 53 (SS)**. Estes pinos dão suporte à comunicação SPI por meio da *biblioteca SPI*. Os pinos SPI também estão disponíveis no conector ICSP que é fisicamente compatível com o Uno, Duemilanove e Diecimila (ou outros modelos de Arduino).
- **LED: 13**. Há um LED conectado ao pino digital 13. Quando o pino está em nível lógico HIGH, o LED se acende e quando o pino está nível lógico LOW, o LED fica desligado.
- **TWI (I2C): 20 (SDA) e 21 (SCL)**. Fornecem suporte à comunicação TWI utilizando a *biblioteca Wire*. Note que estes pinos não estão na mesma posição que no Duemilanove ou Diecimila.

O Mega2560 tem 16 entradas analógicas, cada uma das quais é digitalizada com 10 bits de resolução (i.e. 1024 valores diferentes). Por padrão elas medem de 0 a 5 V, embora seja possível mudar o limite superior usando o pino AREF e a função *analogReference()*.

Há ainda pino *reset* que ao receber um nível lógico LOW, reseta o microcontrolador. Esse pino é tipicamente usado para adicionar um botão de reset em *shields* (placas que adicionam recursos ao Arduino.

2.4 Comunicação

O Arduino Mega 2560 possui várias possibilidades de comunicação com um computador, com outro Arduino ou outros microcontroladores. O ATmega2560 fornece quatro portas de comunicação serial (UARTs) usando níveis TTL (5V). Um chip FTDI FT232RL direciona uma destas portas para a conexão USB e os drivers FTDI (que acompanham o software do Arduino) criam uma porta serial virtual no computador, que pode ser utilizada por qualquer software. A plataforma de desenvolvimentodo Arduino inclui um monitor serial que permite que caracteres sejam enviados da placa Arduino e para a placa Arduino. Os LEDs RX e RT piscarão enquanto dados estiverem sendo transmitidos pelo chip FTDI e pela conexão USB com o computador (mas não para comunicação serial nos pinos 0 e 1).

A biblioteca *SoftwareSerial* permite uma comunicação serial através de qualquer um dos pinos digitais do Mega 2560.

O ATmega2560 também fornece suporte para comunicação I2C (TWI) e SPI. O software Arduino inclui uma biblioteca (*Wire*) para simplificar o uso do barramento I2C. Para utilizar a comunicação SPI deve-se verificar o *datasheet* do ATmega2560.

2.5 Reset Automático (Software)

Em vez de necessitar de um pressionamento físico do botão de reset antes de um upload, o Arduino Mega 2560 é projetado de modo a permitir que o reset seja feito pelo software executado em um computador conectado. Uma das linhas de controle do fluxo por hardware (DTR) do ATmega8U2 é conectada diretamente à linha de reset do ATmega2560 através de um capacitor de 100 nF. Quando esta linha é colocada em nível lógico baixo, a linha de *reset* vai para níve baixo por um tempo suficiente para resetar o microcontrolador. O software Aduino utiliza esta capacidade para possibilitar que novos códigos sejam enviados simplesmente clicando no botão de *upload* do ambiente de desenvolvimento do Arduino.

Esta configuração tem outras implicações. Quando o Mega2560 está conectado via USB a um computador com sistema operacional Mac OS X ou Linux, ele é resetado cada vez que uma

conexão é feita com o software (via USB). Durante o próximo meio segundo, aproximadamente, o *bootloader* é executado no Mega2560. Embora seja programado para ignorar dados corrompidos (i.e. qualquer coisa que não seja o *upload* de um novo código), ele irá interceptar os primeiros bytes de dados enviados à placa depois que a nova conexão é estabelecida. Se um programa rodando no Arduino recebe uma pré-configuração ou outros dados assim que é iniciado, deve-se certificar de que o software com o qual ele se comunica espera meio segundo depois que a conexão foi estabelecida antes de começar a enviar os dados.

O Mega 2560 tem uma trilha que pode ser cortada para desabilitar o *reset* automático. Esta trilha pode ser reconectada por solda para reabilitar esta funcionalidade. Esta trilha tem a identificação "RESET-EN". Também é possível desabilitar o *reset* automático conectando-se um resistor de 110Ω entre a linha de *reset* e 5 V.

2.6 Proteção contra sobrecorrente via USB

O Arduino Mega2560 possui um fusível resetável que protege a porta USB do computador contra curto-circuitos e sobrecorrentes. Apesar de muitos computadores possuírem sua própria proteção interna, o fusível resetável proporciona um grau extra de segurança. Se mais de 500 mA forem drenados ou aplicados na porta USB, o fusível automaticamente abrirá o circuito até que o curto ou a sobrecarga sejam removidos.

2.7 Características Físicas e Compatibilidade com *Shields*

Além das funcionalidades presentes no Arduino Mega 2560, pode-se adicionar kits acessórios diretamente sobre o Arduino, a fim de se obter outras características e recursos tecnológicos não disponíveis no Arduino. A Figura 2 apresenta um kit acessório (*shield*) que implementa o protocolo para comunicação *wireless* ZigBee acoplado ao kit Arduino.

As dimensões máximas de comprimento e largura da placa Mega2560 são 4,0"(101,60 mm) e 2,1"(53,34 mm), respectivamente, com o conector USB e o *jack* de alimentação ultrapassando um pouco essas dimensões. Três furos para montagem com parafusos permitem montar a placa numa superfície ou caixa. Nota-se que a distância entre os pinos 7 e 8 de entrada e saída digital é de 0,16"e não 0,10"como entre os outros pinos .

O Mega2560 é projetado para ser compatível com a maioria dos *shields* construídos para o Uno Diecimila ou Duemilanove. Os pinos de entrada e saída digital 0-13 (e adjacentes AREF e GND), as entradas analógicas 0-5, o conector Power e o ICSP estão todos em posições equivalentes. Além disso, a UART principal (porta serial) está localizada nos mesmos pinos (0 e 1), bem com as interrupções 0 e 1 (pinos 2 e 3, respectivamente). O SPI está disponível através do conector ICSP no Arduino Mega 2560 e no Duemilanove/Diecimila. Nota-se que o I2C não está localizado nos

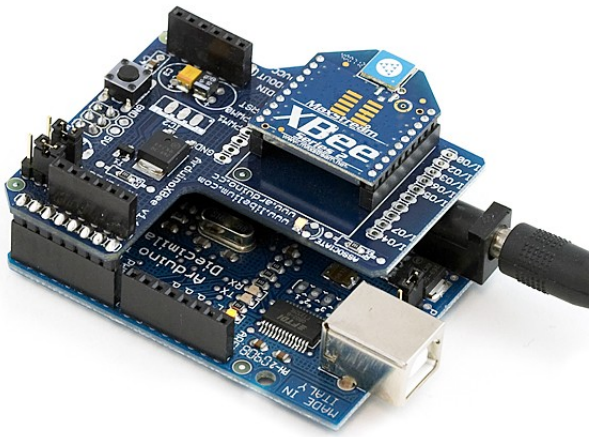

Figura 2: Exemplo de utilização do Arduino com kit acessório Xbee.

mesmos pinos no Mega2560 (20 e 21) e no Duemilanove/Diecimila (entradas analógicas 4 e 5).

3 Desenvolvimento de Programas para o Arduino

3.1 Ambiente de Desenvolvimento

O ambiente de desenvolvimento do Arduino contém um editor de texto para escrita do código, uma área de mensagem, uma área de controle de informações, uma barra de ferramentas com botões para funções comuns e um conjunto de menus. Esse ambiente se conecta ao hardware Arduino para transformar os programas e se comunicar com eles. Os programas escritos usando o ambiente de desenvolvimento Arduino são chamados de *sketches*. O ambiente de desenvolvimento foi desenvolvido em Java e é derivado do ambiente de desenvolvimento para a linguagem *Processing*.

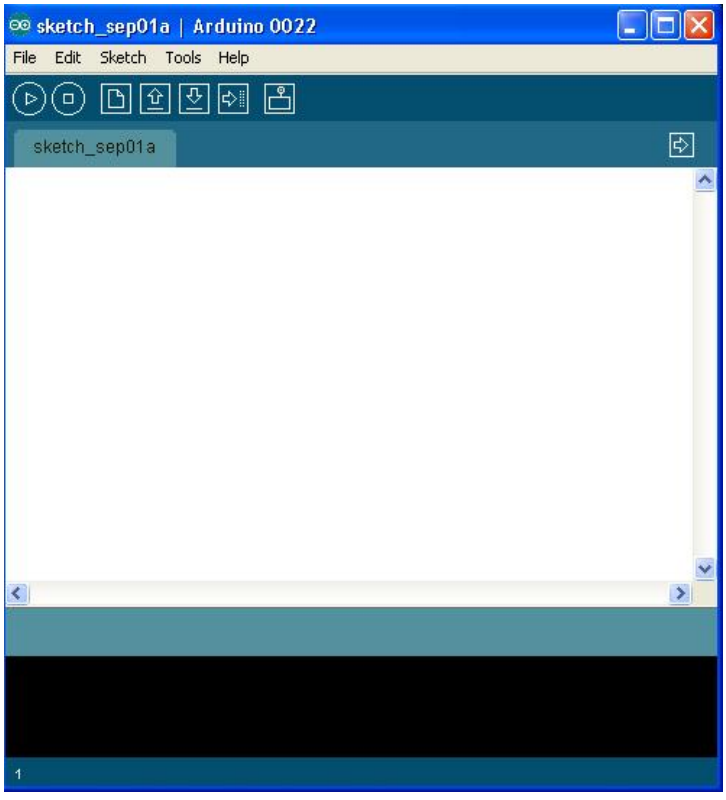

Figura 3: Ambiente de desenvolvimento (IDE) do Arduino

A biblioteca "*Wiring*" disponibilizada junto com o ambiente de desenvolvimento do Arduino possibilita que os programas sejam organizados através de duas funções, embora sejam programas C/C++. Essas duas funções, obrigatórias em todos os programas escritos, são:

- setup(): função que é executada uma única vez no início do programa e é usada para fazer configurações.
- loop(): função que é executada repetidamente até que o kit seja desligado.

O ambiente Arduino usa o conjunto de ferramentas de compilação **gnu C** e a biblioteca **AVR libc** para compilar programas. Usa ainda a ferramenta **avrdude** para carregar programas no Arduino.

Principais comandos disponíveis através de botões:

(a) **Verify/Compile** - Verifica se o código tem erros

(b) **Stop** - Para o monitor serial ou interrompe processos que foram iniciados pelos outros botões.

(c) **New** - Cria um novo sketch.

(d) **Open** - Mostra uma lista de todos os sketches salvos e abre o que for selecionado.

(e) **Save** - Salva o sketch.

(f) **Upload to I/O Board** - Compila o código e transfere para o Arduino.

(g) **Serial Monitor** - Mostra a informação enviada pelo Arduino para o computador.

Comandos adicionais são encontrados através dos menus: File, Edit, Sketch, Tools, Help. As funções disponíveis nos menus *File*, *Edit* e *Help* são semelhantes às de outros programas bem conhecidos e, por isso, não serão detalhadas aqui.

menu Sketch

- **Verify/Compile** - Verifica se o código tem erros
- **Import Library** - Adiciona bibliotecas ao programa
- **Show sketchfolder** - Abre a pasta onde o programa está salvo
- **Add File...** - Adiciona um arquivo fonte ao programa. O novo arquivo aparece em uma nova aba

menu Tools

- **Auto format** - Formata o código para uma melhor leitura, alinhando as chaves e indentando seu conteúdo.
- **Board** - Seleciona o modelo da placa Arduino utilizada no projeto.
- **Serial Port** - Mostra todas as portas seriais disponíveis no computador.
- **Burn Bootloader** - Permite gravar um *bootloader* no Arduino.

3.2 Programando para o Arduino: Conceitos e Sintaxe da Linguagem de Programação

Como já exposto, a linguagem de programação do Arduino é baseada nas linguagens C/C++, preservando sua sintaxe na declaração de variáveis, na utilização de operadores, na manipulação de vetores, na conservação de estruturas, sendo também *case sensitive* (diferencia letras minúsculas e maiúsculas) Contudo, ao invés de uma função *main()*, o programa necessita de duas funções elementares: *setup()* e *loop()*.

Pode-se dividir a linguagem de programação para Arduino em três partes principais: as variáves e constantes, as estruturas e, por último, as funções.

3.2.1 Elementos de Sintaxe

- ; *(ponto e vírgula)* sinaliza a separaração e/ou finalização de instruções.
  Sintaxe:

```
instrução;
```

- {} *(chaves)* é utilizada para delimitar um bloco de instruções referente a uma função (setup,loop...), a um laço (for, while,...), ou ainda, a uma sentença condicional (if...else, switch case...).
  Sintaxe:

```
função/laço/sentença_condicional {
  instruções;
}
```

- // *(linhas de comentários simples)* O conteúdo inserido após // até o final dessa linha, é ignorado pelo compilador e considerado um comentário. O propósito dos comentários é ajudar a entender (ou relembrar) como o programa funciona.
  Sintaxe:

```
instrução;  // aqui, todo comentário é ignorado pelo compilador
```

- /* */ *(bloco de comentário)* tem por finalidade comentar trechos de código no programa. Assim como as *linhas de comentários simples*, o bloco de comentário geralmente é usado

para resumir o funcionamento do que o programa faz e para comentar e descrever funções. Todo conteúdo inserido entre /* */ também é ignorado pelo compilador.
Sintaxe:

```
/* Use o bloco de comentário para descrever,
comentar ou resumir funções
e a funcionalidade do programa.
*/
```

- #**define** permite dar um nome a uma constante antes que o programa seja compilado. Constantes definidas no Arduino não ocupam espaço na memória. O compilador substitui referências a estas constantes pelo valor definido. Não deve-se usar ponto-e-vírgula *(;)* após a declaração *#define* e nem inserir o operador de atribuição " = ", pois isso gerará erros na compilação.

  Sintaxe:

  ```
  #define nome_constante constante
  ```

- #**include** é usado para incluir outras bibliotecas no programa. Isto permite acessar um grande número de bibliotecas padrão da linguagem C (de funções pré-definidas), e também as bibliotecas desenvolvidas especificamente para o Arduino. De modo similar ao *#define*, não deve-se usar ponto-e-vírgula *(;)* no final da sentença.
  Sintaxe:

  ```
  #include <nome_da_biblioteca.h>
  ou
  #include "nome_da_biblioteca.h"
  ```

Exemplos de programas com a utilização dessa sintaxe básica serão apresentados posteriormente nesta apostila.

3.2.2 Setup e Loop

Todo programa criado para o Arduino deve obrigatoriamente possuir duas funções para que o programa funcione corretamente: a função *setup( )* e a função *loop( )*. Essas duas funções não utilizam parâmetros de entrada e são declaradas como *void*. Não é necessário invocar a função *setup()* ou a função *loop()*. Ao compilar um programa para o Arduino, o compilador irá, automaticamente, inserir uma função *main* que invocará ambas as funções.

setup( )

A função *setup* é utilizada para inicializar variáveis, configurar o modo dos pinos e incluir bibliotecas. Esta função é executada automaticamente uma única vez, assim que o kit Arduino é ligado ou resetado.
Sintaxe:

```
void setup()
{
    .
    :
}
```

loop( )

A função *loop* faz exatamente o que seu nome sugere: entra em looping (executa sempre o mesmo bloco de código), permitindo ao programa executar as instruções que estão dentro desta função. A função *loop()* deve ser declarada após a função *setup()*
Sintaxe:

```
void loop()
{
    .
    :
}
```

Variáveis e Constantes

3.2.3 Variáveis

Variáveis são áreas de memória, acessíveis por nomes, que pode-se usar em programas para armazenar valores, como por exemplo, a leitura de um sensor conectado em uma entrada analógica.

A seguir aparecem exemplos de alguns trechos de código.

3.2.4 Tipos de dados

As variáveis podem ser de vários tipos:

- **boolean** Variáveis booleanas podem ter apenas dois valores: *true* (verdadeiro) ou *false* (falso).

  Sintaxe:

  ```
  boolean variável = valor
  // valor = true ou false
  ```

  Exemplo:

  ```
  boolean teste = false;
  ...
  if (teste == true)
    i++;
  ...
  ```

- **byte** Armazena um número de 8 bits sem sinal (*unsigned*), de 0 a 255.

  Sintaxe:

  ```
  byte variavel = valor;
  ```

  Exemplo:

  ```
  byte x = 1;
  byte b = B10010;
  // B indica o formato binário
  // B10010 = 18 decimal
  ```

- **char** É um tipo de variável que ocupa 1 byte de memória e armazena o código ASCII de um caractere. Caracteres literais são escritos com ' '(aspas simples) como: 'N'. Para cadeia de caracteres utiliza-se " " (aspas duplas), como : "ABC".

Sintaxe:

```
char variavel  = 'caracater';
char variavel = "frase";
```

Exemplo:

```
char mychar  = 'N';
// mychar recebe o valor 78,
// Correspondente ao
// caracter 'N' segundo a tabela ASCII
```

- **int** Inteiro é o principal tipo de dado para armazenamento numérico capaz de armazenar números de 2 bytes. Isto abrange a faixa de -32.768 a 32.767.

Sintaxe:

```
int var = valor;
```

Exemplo:

```
int ledPin = 13; //ledPin recebe 13
int x = -150;    //x recebe -150
```

- **unsigned int** Inteiros sem sinal permitem armazenar valores de 2 bytes. Entretanto, ao invés de armazenar números negativos, armazenam somente valores positivos abrangendo a faixa de 0 a 65.535. A diferença entre inteiros e inteiros sem sinal está no modo como o bit mais significativo é interpretado. No Arduino, o tipo int (que é com sinal) considera que, se o bit mais significativo é 1, o número é interpretado como negativo. Os tipos com sinal representam números usando a técnica chamada **complemento de 2**.

Sintaxe:

```
unsigned int var = val;
```

Exemplo:

```
unsigned int ledPin = 13;
```

- **long** Variáveis do tipo Long têm um tamanho ampliado para armazenamento de números, sendo capazes de armazenar 32 bits (4 bytes), de -2.147.483,648 a 2.147.483.647.

Sintaxe:

```
long variavel = valor;
```

Exemplo:

```
long exemplo  = -1500000000
long exemplo2 = 2003060000
```

- **unsigned long** Longs sem sinal são variáveis de tamanho ampliado para armazenamento numérico. Diferente do tipo long padrão, esse tipo de dado não armazena números negativos, abrangendo a faixa de 0 a 4.294.967.295.

  Sintaxe:

  ```
  unsigned long variavel = valor;
  ```

  Exemplo:

  ```
  unsigned long var = 3000000000;
  ```

- **float** Tipo de variável para números de "ponto flutuante" que possibilitam representar valores reais muito pequenos e muito grandes. Números do tipo float utilizam 32 bits e abrangem a faixa de -3,4028235E+38 a 3,4028235E+38.

  Sintaxe:

  ```
  floar var = val;
  ```

  Exemplo:

  ```
  float sensorCalbrate = 1.117;
  ```

- **double** Número de ponto flutuante de precisão dupla. A implementação do double no Arduino é, atualmente, a mesma do float, sem ganho de precisão, ocupando 4 bytes também.

  Sintaxe:

  ```
  double var = val;
  ```

  Exemplo:

  ```
  double x = 1.117;
  ```

- **array** Um array (vetor) é uma coleção de variáveis do mesmo tipo que são acessadas com um índice numérico. Sendo a primeira posição de um vetor V a de índice 0 *(V[0])* e a última de índice $n-1$ *(V[n-1])* para um vetor de n elementos.
  Um vetor também pode ser multidimensional *(matriz)*, podendo ser acessado como: $V[m][n]$, tendo $m \times n$ posições. Assim, tem-se a primeira posição de V com índice 0,0 *(V[0][0])* e a última sendo $m-1,\ n-1$ *($V[m-1][n-1]$)*.
  Um *array* pode ser declarado sem especificar o seu tamanho.

Sintaxe:

```
tipo_variável var[];
tipo_variável var[] = valor;
tipo_variável var[índice] = valor;
tipo_variável var[][];
tipo_variável var[][índice] = valor;
tipo_variável var[índ][índ] = valor;
```

Exemplo:

```
int  var[6];
int  myvetor[] = {2, 4, 8, 3, 6};
int  vetor[6] = {2, 4, -8, 3, 2};
char message[6] = "hello";
int  v[2][3] = {0 1 7
                3 1 0};
int  A[2][4] = {{2 7}
                {3 2 5 6}};
```

- **string** Strings são representadas como um vetor do tipo char e terminadas por *null*(nulo). Por serem terminadas em *null* (código ASCII 0), permitem às funções (como Serial.print()) saber onde está o final da string. De outro modo elas continuariam lendo os bytes subsequentes da memória que de fato não pertencem à string. Isto significa que uma string deve ter espaço para um caractere a mais do que o texto que ela contém.

Sintaxe:

```
tipo_variável var[índice] = valor;
```

Exemplo:

```
char Str1[15];
char Str2[5] = {'m','e','g','a'};
char Str3[5] = {'m','e','g','a','\0'};
char Str4[ ] = "arduino";
char Str5[5] = "mega";
```

Como apresentado no exemplo anterior, Str2 e Str5 precisam ter 5 caracteres, embora "mega" tenha apenas 4. A última posição é automaticamente preenchida com o caractere *null*. Str4 terá o tamanho determinado automaticamente como 8 caracteres, um extra para o *null*. Na Str3 foi incluido explicitamente o caractere *null* (escrito como "\0"). Na Str1 definiu-se uma string com 15 posições não inicializadas, lembrando que a última posição correspondente à posição 15 é reservada para o caracter *null*.

- **void** A palavra chave *void* é usada apenas em declarações de funções. Ela indica que a função não deve enviar nenhuma informação de retorno à função que a chamou. Como exemplo de funções declaradas com retorno *void* tem-se as funções *setup* e *loop*.

Sintaxe:

```
void nome_função()
void nome_função(parametros)
```

Exemplo:

```
void setup()
{
  .
  :
}

void loop()
{
  .
  :
}
```

3.2.5 Constantes

Constantes são nomes com valores pré-definidos e com significados específicos que não podem ser alterados na execução do programa. Ajudam a deixar o programa mais facilmente legível. A linguagem de programação para o Arduino oferece algumas constantes acessíveis aos usuários.

Constantes booleanas (verdadeiro e falso)

Há duas constantes usadas para representar verdadeiro ou falsona linguagem Arduino: true (verdadeiro), e false (falso).

- **false** false é a mais simples das duas e é definida como 0 (zero).
- **true** true é frequentemente definida como 1, o que é correto, mas true tem uma definição mais ampla. Qualquer inteiro que não é zero é TRUE, num modo booleano. Assim, -1, 2 e -200, 70 são todos definidos como true.

HIGH e LOW

Quando se está lendo ou escrevendo em um pino digital há apenas dois valores que um pino pode ter: HIGH *(alto)* e LOW *(baixo)*.

- **HIGH** O significado de HIGH (em referência a um pino) pode variar um pouco dependendo se este pino é uma entrada (INPUT) ou saída (OUTPUT). Quando um pino é configurado como INPUT com a função pinMode, e lido com a função digitalRead, o microcontrolador considera como HIGH se a tensão for de 3 V ou mais. Um pino também pode ser configurado como um INPUT, e posteriormente receber um HIGH com um digitalWrite, isto vai "levantar" o resistor interno de 20 KOhms que vai manter a leitura do pino como HIGH a não ser que ela seja alterada para LOW por um circuito externo. Quando um pino é configurado como OUTPUT, e definido como HIGH com o digitalWrite, ele fica com 5 V. Neste estado ele pode enviar corrente para, por exemplo, acender um LED que está conectado com um resistor em série ao GND, ou a outro pino configurado como OUTPUT e definido como LOW.
- **LOW** O significado de LOW também pode variar dependendo do pino ser definido como INPUT ou OUTPUT. Quando um pino é configurado como INPUT com a função pinMode, e lido com a função digitalRead, o microcontrolador considera como LOW se a tensão for de 2 V ou menos. Quando um pino é configurado como OUTPUT, e definido como LOW, ele fica com 0 V. Neste estado ele pode "drenar" corrente para, por exemplo, acender um

LED que está conectado com um resistor em série ao +5 Volts, ou a outro pino configurado como OUTPUT e definido como HIGH.

INPUT e OUTPUT

Pinos digitais podem ser configurados como INPUT e como OUTPUT. Mudar um pino de INPUT para OUTPUT com *pinMode()* muda drasticamente o seu comportamento elétrico.

- **INPUT** Os pinos do Arduino (Atmega) configurados como INPUT com a função *pinMode()* estão em um estado de alta impedância. Pinos de entrada são usados para ler um sensor mas não para energizar um LED.
- **OUTPUT** Pinos configurados como OUTPUT com a função *pinMode()* estão em um estado de baixa impedância. Isto significa que eles podem fornecer grandes quantidades de corrente para outros circuitos. Os pinos do Atmega podem fornecer (corrente positiva) ou drenar (corrente negativa) até 40 mA (milliamperes) de/para outros dispositivos ou circuitos. Isto faz com que eles sejam úteis para energizar um LED mas inapropriados para a leitura de sensores. Pinos configurados como OUTPUT também podem ser danificados ou destruidos por curto-circuitos com o GND ou com outros pontos de 5 Volts. A quantidade de corrente fornecida por um pino do Atmega também não é suficiente para ativar muitos relês e motores e, neste caso, algum circuito de interface será necessário.

3.2.6 Conversão

converte de	Para	Sintaxe
boolean	char	char(variavel)
int		
long		
float		
double		
char	int	int(variavel)
boolean		
long		
float		
double		
char	float	float(variavel)
boolean		
long		
int		
double		
char	double	double(variavel)
boolean		
long		
int		
float		
char	boolean	boolean(variavel)
float		
long		

int		
double		
char	long	long(variavel)
float		
boolean		
int		
double		

3.2.7 Estrutura de Controle

- **if:** estrutura utilizada com a finalidade de verificar se uma condição é verdadeira. Em caso afirmativo, executa-se um bloco do código com algumas instruções. Caso contrário, o programa não executa o bloco de instruções e pula o bloco referente a essa estrutura.

Sintaxe:

```
if (condição)
{
bloco de instrução;
}
```

Exemplo:

```
if (x > 120)
 int y = 60;
```

- **if...else:** permite um controle maior sobre o fluxo de código do que a sentença *if* básica. Quando usa-se a estrutura *if...else* garante-se que uma das duas declarações será executada. Nunca serão executadas as duas ou nenhuma delas. Caso a condição do *if* seja satisfeita, executa-se o bloco de instruções referente ao *if*, caso contrário, executa-se obrigatoriamente o bloco de instruções do *else*.

Sintaxe:

```
If (condição) {
bloco de instrução 1
}
else{
bloco de instrução 2
}
```

Exemplo:

```
if (x <= 500)
  int y = 35;
else{
  int y = 50 + x;
  x = 500;
}
```

- **switch case:** permite construir uma lista de "casos" dentro de um bloco delimitado por chaves. O programa verifica cada caso com a variável de teste e executa determinado bloco de instrução se encontrar um valor idêntico. A estrutura *switch case* é mais flexível que a estrutura *if...else* já que pode-se determinar se a estrutura switch deve continuar verificando se há valores idênticos na lista dos "casos" após encontrar um valor idêntico, ou não. Deve-se utilizar a sentença *break* após a execução do bloco de código selecionado por um dos "casos". Nessa situação, se uma sentença *break* é encontrada, a execução do

programa "sai" do bloco de código e vai para a próxima instrução após o bloco **switch case**. Se nenhuma condição for satisfeita o código que está no *default* é executado. O uso do *default* ou seja, de uma instrução padrão, é opcional no programa.

Sintaxe:

```
switch (variável) {
  case 1:
    instrução p/ quando variável == 1
    break;
 case 2:
    instrução p/ quando variáevel == 2
    break;
 default:
    instrução padrão
}
```

Exemplo:

```
switch (x) {
  case 1:
      y = 100;
      break;
  case 2:
      y = 158;
      break;
  default:
      y = 0;
}
```

- **for:** é utilizado para repetir um bloco de código delimitado por chaves. Um contador com incremento/decremento normalmente é usado para controlar e finalizar o laço. A sentença *for* é útil para qualquer operação repetitiva.
Há três partes no cabeçalho de um *for*:

for (inicialização; condição; incremento)

A *inicialização* ocorre primeiro e apenas uma vez. Cada vez que o laço é executado, a *condição* é verificada; se ela for verdadeira, o bloco de código é executado. Em seguida, o *incremento* é realizado, e então a condição é testada novamente. Quando a condição se torna falsa o laço termina.

Sintaxe:

```
for(inicializa;condição;incremento)
{
  bloco de instruções;
}
```

Exemplo:

```
for (int i=0; i <= 255; i++){
    char str[i] = i;
       .
       :
}
```

- **while:** permite executar um bloco de código entre chaves repetidamente por inúmeras vezes até que a *(condição)* se torne falsa. Essa condição é uma sentença booleana em C que pode ser verificada como verdadeira ou falsa.

Sintaxe:

```
while(condição){
  bloco de instruções;
}
```

Exemplo:

```
int i = 0;
while(i < 51){
      .
      :
   i++;
}
```

- **do...while:** funciona da mesma maneira que o *while*, com a exceção de que agora a condição é testada no final do bloco de código. Enquanto no *while*, se a condição for falsa, o bloco de código não será executado, no *do...while* ele sempre será executado pelo menos uma vez.

Sintaxe:

```
do
{
  bloco de instruções;
} while (condição);
```

Exemplo:

```
int x= 20;
do {
    .
    :
  x--;
} while (x > 0);
```

- **continue:** é usado para saltar porções de código em comandos como *for do...while*, *while*. Ele força com que o código avance até o teste da condição, saltando todo o resto.

Sintaxe usando o while:

```
while (condição){
   bloco de instruções;
   if (condição)
       continue;
   bloco de instruções;
}
```

Exemplo de trecho de código usando for:

```
for (x=0;x<255;x++){
  if(x>40 && x<120)
      continue;
      .
      :
}
```

- **break:** é utilizado para sair de um laço *do...while*, *for*, *while* ou *switch case*, se sobrepondo à condição normal de verificação.

Sintaxe usando do...while:

```
do{
  bloco de instruções;
  if (condição)
    bloco de instruções;
    break;
} while (condição);
```

Exemplo de trecho de código usando while:

```
x=1;
while(x<255){
   y = 12/x;
   if (y < x){
     x = 0;
     break;
  }
  x++;
}
```

- **return:** finaliza uma função e retorna um valor, se necessário. Esse valor pode ser uma variável ou uma constante.

Sintaxe:

```
return;
ou
return valor;
```

Exemplo:

```
  if (x > 255)
      return 0;
  else
      return 1;
```

3.2.8 Operadores de Comparação

Os operadores de comparação, como o próprio nome sugere, permitem que se compare dois valores. Em qualquer das expressões na Tabela 3, o valor retornado sempre será um valor booleano. É possível realizar comparações entre números, caracteres e booleanos. Quando se utiliza um caractere na comparação, o código *ASCII* desse caractere é considerado. Para comparar um *array* com outro tipo de dado, deve-se indicar uma posição do *array* para ser comparado.

Tabela 3: Operadores de Comparação

Operando Direito	Operador	Operando Esquerdo	Retorno
boolean	= =	boolean	boolean
int	!=	int	
float	<	float	
double	>	double	
char	<=	char	
array[ ]	>=	array[ ]	

3.2.9 Operador de Atribuição

O operador de atribuição (ou operador de designação) armazena o valor à direita do sinal de igual na variável que está à esquerda desse sinal. Esse operador também indica ao microcontrolador para calcular o valor da expressão à direita e armazenar este valor na variável que está à esquerda.

```
x = y; (a variável x armazena o valor de y)

a = b + c; (calcula o resultado da soma e coloca na variável a)
```

3.2.10 Operadores Aritméticos

Os operadores aritméticos são usados para realizar operações matemáticas.

Tabela 4: Operadores Aritméticos com **int**

Operando Direito	Operador	Operando Esquerdo	Retorno
int	+	int double float char	int
	-		
	*		
	/		
	%	int char	

Tabela 5: Operadores Aritméticos com **char** / **array[ ]**

Operando Direito	Operador	Operando Esquerdo	Retorno
char	+	int double float char	char
	-		
	*		
	/		
	%	int char	

Tabela 6: Operadores Aritméticos com **double** / **float**

Operando Direito	Operador	Operando Esquerdo	Retorno
float	+	int	float
	-	double	
double	*	float	double
	/	char	

3.2.11 Operadores Booleanos

Os operadores booleanos (ou operadores lógicos) são geralmente usados dentro de uma condição *if* ou *while*. Em geral, os operandos da expressão podem ser números, expressões relacionais e sempre retornam como resposta um valor lógico: Verdadeiro (1) ou Falso (0).

- **&&** *(e)* exige que os dois operandos sejam verdadeiros para ser verdade, ou seja, a primeira condição "e" a segunda devem ser verdadeiras;
- || *(ou)* para ser verdadeiro, basta que um dos operando seja verdade, ou seja, se a primeira condição "ou" a segunda "ou" ambas é(são) verdadeira(s), então o resultado é verdadeiro;
- **!** *(não)* é verdadeiro apenas quando o operando for falso.

3.2.12 Operadores de Bits

Os operadores de bits realizam operações ao nível de bits das variáveis.
Esses operadores operam somente em dados do tipo char ou int.

- & *(operador AND bit a bit)* é usado entre duas variáveis/constantes inteiras. Ele realiza uma operação entre cada bit de cada variável de acordo com a seguinte regra: se os dois bits de entrada forem 1, o resultado da operação também é 1, caso contrário é 0.
  Exemplo:

```
0  0  1  1   a
0  1  0  1   b
----------
0  0  0  1  ( a & b)
```

&	0	1
1	0	1
0	0	0

- | *(operador OR bit a bit)* realiza operações com cada bit de duas variáveis conforme a seguinte regra: o resultado da operação é 1 se um dos bits de entrada for 1, caso contrário é 0.
  Exemplo:

```
0  0  1  1   c
0  1  0  1   d
----------
0  1  1  1   ( c | d )
```

|	0	1
1	1	1
0	0	1

- ^ *(operador XOR bit a bit)* Conhecido como *Exclusive or* (ou exclusivo), esse operador realiza uma operação entre cada bit de cada variável de acordo com a seguinte regra: se os dois bits de entrada forem diferentes, o resultado desta operação é 1 , caso contrário, retorna 0.
  Exemplo:

```
0  0  1  1   e
0  1  0  1   f
----------
0  1  1  0   ( e ^ f )
```

^	0	1
1	1	0
0	0	1

- ~ *(operador de bits NOT)* diferente dos operadores AND, OR e XOR, este operador é aplicado apenas sobre um operando, retornando o valor inverso de cada bit.
  Exemplo:

```
0  1    g
----
1  0   (~g)
```

~	
0	1
1	0

- $<<$ *(deslocamento à esquerda)* Desloca para a esquerda os bits do operando esquerdo conforme o valor dado pelo operando direito.

  Exemplo:

```
int a = 3;
int x = a << 2;

   0 0 0 0 1 1   a
   0 1 1 0 0 0   a << 3
```

byte	$<<$	retorno
00000001	2	00000100
00000101	3	00101000

- $>>$ *(deslocamento à direita)* Desloca, para a direita, os bits do operando esquerdo conforme o valor dado pelo operando direito.

  Exemplo:

```
int b = 40;
int y = b >> 3;

  0 1 0 1 0 0 0   b
  0 0 0 0 1 0 1   b >> 3
```

byte	$>>$	retorno
00001000	2	00000010
00001001	3	00000001

3.2.13 Operadores Compostos

Os operadores compostos consistem em um recurso de escrita reduzida provido pela linguagem C, havendo sempre a possibilidade de obter-se o resultado equivalente através do uso de operadores simples.

Incremento e Decremento

Os incrementos (++) e decrementos (- -) podem ser colocados antes ou depois da variável a ser modificada. Se inseridos antes, modificam o valor antes da expressão ser usada e, se inseridos depois, modificam depois do uso.

- ++ *(incremento)* aumenta o valor de variáveis em uma unidade;
  Exemplo:

```
int x = 2;
int var = ++x;
```

o valor de var será 3 e o de x será 3.

```
x = 2;
var = x++;
```

o valor de var será 2 e o de x será 3 .

- - - *(decremento)* diminui o valor de variáveis em uma unidade;
  Exemplo:

```
int x = 7;
int var = --x;
```

o valor de var será 6 e o de x será 6.

```
x = 7;
var = x--;
```

o valor de var será 7 e o de x será 6 .

- += *(adição composta)* realiza a adição de uma variável com outra constante ou variável.
  Exemplo:

```
x += y;
```

equivale à expressão x = x + y

```
x = 2;
x += 4;
```

x passa a valer 6

- -= *(subtração composta)* realiza a subtração de uma variável com outra constante ou variável.
  Exemplo:

```
x -= y;
```

equivale à expressão x = x - y

```
x = 7;
x -= 4;
```

x passa a valer 3

- *= *(multiplicação composta)* realiza a multiplicação de uma variável com outra constante ou variável.
  Exemplo:

```
x *= y;
```

equivale à expressão x = x * y

```
x = 8;
x *= 2;
```

x passa a valer 16

- /= *(divisão composta)* realiza a divisão de uma variável com outra constante ou variável.
  Exemplo:

```
x /= y;
```

equivale à expressão x = x / y

```
x = 10;
x /= 2;
```

x passa a valer 5

Funções

3.2.14 Entrada e saída digital

- **pinMode( )** Configura o pino especificado para que se comporte como uma entrada ou como uma saída. Deve-se informar o número do pino que deseja-se configurar e em seguida se o pino será uma entrada (INPUT) ou uma saída (OUTPUT).
  Sintaxe:

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

- **digitalWrite( )** Escreve um valor HIGH ou LOW em um pino digital. Se o pino foi configurado como uma saída, sua tensão será: 5V para HIGH e 0V para LOW. Se o pino está configurado como uma entrada, HIGH levantará o resistor interno de 20KOhms e LOW rebaixará o resistor.
  Sintaxe:

```
digitalWrite(pino, valor);
```

- **digitalRead( )** Lê o valor de um pino digital especificado e retorna um valor HIGH ou LOW.
  Sintaxe:

```
int digitalRead(pino);
```

```
/* Exemplo de função sobre de Entrada e Saída Digital */

int ledPin = 13;  // LED conectado ao pino digital 13
int inPin = 7;    // botão conectado ao pino digital 7
int val = 0;      // variável para armazenar o valor lido

void setup()
{
  pinMode(ledPin, OUTPUT);   // determina o pino digital 13 como uma saída
  pinMode(inPin, INPUT);      // determina o pino digital 7 como uma entrada
}

void loop()
{
  digitalWrite(ledPin, HIGH);   // acende o LED
  val = digitalRead(inPin);     // lê o pino de entrada
  digitalWrite(ledPin, val);    // acende o LED de acordo com o pino de entrada
}
```

Essa função transfere para o pino 13, o valor lido no pino 7 que é uma entrada.

3.2.15 Entrada e saída analógica

- **analogWrite( ) - PWM** (*Pulse Width Modulation* ou Modulação por Largura de Pulso) é um método para obter sinais analógicos com sinais digitais. Essa função, basicamente, define o valor de um sinal analógico. Ela pode ser usada para acender um LED variando o seu brilho, ou controlar um motor com velocidade variável. Depois de realizar um analogWrite(), o pino gera uma onda quadrada estável com o ciclo de rendimento especificado até que um analogWrite(), um digitalRead() ou um digitalWrite() seja usado no mesmo pino.

  Em kits Arduino com o chip ATmega168, esta função está disponível nos pinos 3,5,6,9,10 e 11. Kits Arduino mais antigos com um ATmega8 suportam o analogWrite() apenas nos pinos 9,10 e 11. As saídas PWM geradas pelos pinos 5 e 6 terão rendimento de ciclo acima do esperado. Isto se deve às interações com as funções **millis()** e **delay()**, que compartilham o mesmo temporizador interno usado para gerar as saídas PWM.

  Para usar esta função deve-se informar o pino ao qual deseja escrever e em seguida informar um valor entre 0 (pino sempre em 0V) e 255 (pino sempre em +5V).

  Sintaxe:

```
analogWrite(pino, valor);
```

- **analogRead( )** Lê o valor de um pino analógico especificado. O kit Arduino contém um conversor analógico-digital de 10 bits com 6 canais. Com isto ele pode digitalizar tensões de entrada entre 0 e 5 Volts, em valores inteiros entre 0 e 1023. Isto permite uma resolução entre leituras de 5 Volts / 1024 ou 0,0049 Volts (4,9 mV) por unidade do valor digitalizado.

  Sintaxe:

```
int analogRead(pino);
```

```
/* Exemplo de função sobre Entrada e Saída Analógica */

  int ledPin = 9;       // LED conectado ao pino digital 9
  int analogPin = 3;    // potenciômetro conectado ao pino analógico 3
  int val = 0;          // variável para armazenar o valor lido

  void setup()
  {
    pinMode(ledPin, OUTPUT);   // pré-determina o pino como saída
  }

  void loop()
  {
    val = analogRead(analogPin);   // lê o pino de entrada
    analogWrite(ledPin, val/4);   //
  }
```

Torna o brilho de um LED proporcional ao valor lido em um potenciômetro.

3.2.16 Entrada e saída avançada

- **pulseIn( )** Lê um pulso (tanto HIGH como LOW) em um determinado pino.

  Por exemplo, se o valor for HIGH, a função pulseIn() espera que o pino tenha o valor HIGH, inicia uma cronometragem, e então espera que o pino vá para LOW e pára essa cronometragem. Por fim, essa função retorna a duração do pulso em microsegundos. Caso nenhum pulso iniciar dentro de um tempo especificado (a determinação desse tempo na função é opcional), **pulseIn( )** retorna 0. Esta função funciona com pulsos entre 10 microsegundos e 3 minutos.

  Sintaxe:

  ```
  pulseIn(pino, valor)
  ou
  pulseIn(pino, valor, tempo)
  ```

  ```
  int pin = 7;
  unsigned long duration;

  void setup()
  {
    pinMode(pin, INPUT);
  }

  void loop()
  {
    duration = pulseIn(pin, HIGH);
  }
  ```

- **shiftOut( )** Envia um byte de cada vez para a saída. Pode começar tanto pelo bit mais significativo (mais à esquerda) quanto pelo menos significativo (mais à direita). Os bits vão sendo escritos um de cada vez em um pino de dados em sincronia com as alterações de um pino de clock que indica que o próximo bit deve ser escrito. Isto é um modo usado para que os microcontroladores se comuniquem com sensores e com outros microcontroladores. Os dois dispositivos mantêm-se sincronizados a velocidades próximas da máxima, desde que ambos compartilhem a mesma linha de clock.

  Nesta função deve ser informado o número referente ao *pino* no qual sairá cada bit (pino de dados). Em seguida, declara-se o número do pino que será alterado quando um novo bit deverá sair no primeiro pino (pino de clock). Depois, informa-se qual é a ordem de envio dos bits. Essa ordem pode ser MSBFIRST (primeiro o mais significativo) ou LSBFIRST (primeiro o menos significativo). Por último, declara-se a informação que será enviada para a saída.

  Obs: O *pino de dados* e o *pino de clock* devem ser declarados como saída (OUTPUT) pela função *pinMode()*.

Sintaxe:

```
shiftOut(pino de dados, pino de clock, ordem, informação);
```

Exemplo:

```
int latchPin = 8;
int clockPin = 12;
int dataPin = 11;

void setup() {
  pinMode(latchPin, OUTPUT);
  pinMode(clockPin, OUTPUT);
  pinMode(dataPin, OUTPUT);
}

void loop() {
  for (int j = 0; j < 256; j++) {
    digitalWrite(latchPin, LOW);
    shiftOut(dataPin, clockPin, LSBFIRST, j);
    digitalWrite(latchPin, HIGH);
    delay(1000);
  }
}
```

3.2.17 Tempo

- **millis( )** Retorna o número de milisegundos desde que o kit Arduino começou a executar o programa. Este número voltará a zero depois de aproximamente 50 dias.
  Sintaxe:

```
unsigned long tempo;
void loop
{
      .
      :

  tempo = millis()
}
```

- **delay( )** Suspende a execução do programa pelo tempo (em milisegundos) especificado (1 segundo = 1000 milisegundos).

  Sintaxe:

```
  delay(tempo);
```

- **micros( )** Retorna o número de microsegundos desde que o kit Arduino começou a executar o programa. Este número voltará a zero depois de aproximamente 70 minutos (1 segundo = 1000 milisegundos = 1 000 000 microsegundos).
  Sintaxe:

```
unsigned long tempo;
void loop
{
      .
      :
  tempo = micros();
      .
      :
}
```

Exemplo:

```
/* Este programa mostra uma aplicação das funções millis( ) e delay( )
 * Para usar a função micros( ), basta substituir millis( ) por micros ( ) */

unsigned long time;

void setup(){
  Serial.begin(9600);
}
void loop(){
  Serial.print("Time: ");
  time = millis();
  Serial.println(time); //imprime o tempo desde que o programa começou
  delay(1000);
}
```

- **delayMicroseconds( )** Suspende a execução do programa pelo tempo (em microsegundos) especificado. Atualmente, o maior valor que produzirá uma suspensão precisa é da ordem de 16383. Para suspensões maiores que milhares de microsegundos, deve-se utilizar a função *delay( )*.

Sintaxe:

```
delayMicroseconds(tempo);
```

Exemplo:

```
int outPin = 8;
void setup() {
  pinMode(outPin, OUTPUT);
}
void loop() {
  digitalWrite(outPin, HIGH);
  delayMicroseconds(50);
  digitalWrite(outPin, LOW);
  delayMicroseconds(50);
}
```

3.2.18 Comunicação serial

- **Serial.begin ( )** Ajusta o taxa de transferência em bits por segundo para uma transmissão de dados pelo padrão serial. Para comunicação com um computador utiliza-se uma destas taxas: 300, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 57600, 115200. Pode-se, entretanto, especificar outras velocidades, por exemplo, para comunicação através dos pinos 0 e 1 com um componente que requer uma taxa específica.

  Sintaxe:

```
Serial.begin(taxa);
Serial1.begin(taxa);
Serial12.begin(taxa);
Serial13.begin(taxa);
```

  Exemplo para Arduino Mega:

```
void setup(){

/* Abre a porta serial 1, 2, 3 e 4
 * e ajusta a taxa das portas para
 * 9600 bps, 38400 bps, 19200 bps
 * e 4800 bps respectivamente
 */

  Serial1.begin(9600);
  Serial2.begin(38400);
  Serial3.begin(19200);
  Serial4.begin(4800);
        .
        :

}

void loop() {}
```

- **int Serial.available( )** Retorna o número de bytes (caracteres) disponíveis para leitura no buffer da porta serial. O buffer serial pode armazenar até 128 bytes.

Sintaxe:

```
Serial.available();
```

Exemplo

```
void setup() {
  Serial.begin(9600);
  Serial1.begin(9600);
}

void loop() {
  /* lê na porta 0
   * e envia para a porta 1:
   */
  if (Serial.available()) {
    int inByte = Serial.read();
    Serial1.print(inByte, BYTE);
  }

  /* lê na porta 1 e
   * envia para a porta 0:
   */
  if (Serial1.available()) {
    int inByte = Serial1.read();
    Serial.print(inByte, BYTE);
  }
}
```

- **int Serial.read( )** Retorna o primeiro byte disponível no buffer de entrada da porta serial (ou -1 se não hover dados mo buffer)

Sintaxe

```
variavel = Serial.read( )
```

Exemplo

```
int incomingByte = 0;
// para entrada serial

void setup() {
Serial.begin(9600);
}

void loop() {
// envia dados apenas
//quando recebe dados:
if (Serial.available() > 0) {
// lê o primeiro byte disponível:
incomingByte = Serial.read();

// imprime na tela o byte recebido:
Serial.print("Eu recebi: ");
Serial.println(incomingByte, DEC);
}
}
```

- **Serial.flush( )** Esvazia o *buffer* de entrada da porta serial. De modo geral, esta função apaga todos os dados presentes no *buffer* de entrada no momento de execução da mesma.

Sintaxe:

```
Serial.flush();
```

Exemplo:

```
void setup() {(
  Serial.begin(9600);
}
void loop(){
  Serial.flush();
  /* Apaga o conteúdo
   * do buffer de entrada
   */
   .
   :
}
```

- **Serial.print( )** Envia dados de todos os tipos inteiros, incluindo caracteres, pela porta serial. Ela não funciona com floats, portanto é necessário fazer uma conversão para um tipo inteiro. Em algumas situações é útil multiplicar um float por uma potência de 10 para preservar (ao menos em parte) a informação fracionária. Atente-se para o fato de que os tipos de dados sem sinal, char e byte irão gerar resultados incorretos e atuar como se fossem do tipo de dados com sinal. Este comando pode assumir diversas formas:

Serial.print(valor) sem nenhum formato especificado: imprime o valor como um número decimal em uma string *ASCII*.
Por exemplo:

```
int b = 79;
Serial.print(b);
```

(envia pela porta serial o código *ASCII* do 7 e o código *ASCII* do 9).

Serial.print(valor, DEC): imprime valor como um número decimal em uma string *ASCII*.
Por exemplo:

```
int b = 79;
Serial.print(b, DEC);
```
(imprime a string *ASCII* "79").

Serial.print(valor, HEX): imprime valor como um número hexadecimal em uma string *ASCII*.
Por exemplo:

```
int b = 79;
Serial.print(b, HEX);
```
(imprime a string "4F").

Serial.print(valor, OCT): imprime valor como um número octal em uma string *ASCII*.
Por exemplo:

```
int b = 79;
Serial.print(b, OCT);
```
(imprime a string "117")

Serial.print(valor, BIN): imprime valor como um número binário em uma string *ASCII*.
Por exemplo:

```
int b = 79;
Serial.print(b, BIN);
```
(imprime a string "1001111").

Serial.print(valor, BYTE): imprime valor como um único byte. Por exemplo:

```
int b = 79;
Serial.print(b, BYTE);
```
(envia pela porta serial o valor 79, que será mostrado na tela de um terminal como um caractere "0", pois 79 é o código *ASCII* do "0").

Serial.print(str): se str for uma string ou um array de chars, imprime uma string ASCII.
Por exemplo:

```
Serial.print("Arduino Mega");
```
(imprime a string "Arduino Mega")

Exemplo:

```
int analogValue;

void setup()
{
  serial.begin(9600);
}

void loop()
{
  analogValue = analogRead(0);

  serial.print(analogValue);         // imprime um ASCII decimal - o mesmo que "DEC"
  serial.print("\t");                 // imprime um tab
  serial.print(analogValue, DEC);    // Imprime um valor decimal
  serial.print("\t");                 // imprime um tab
  serial.print(analogValue, HEX);    // imprime um ASCII hexadecimal
  serial.print("\t");                 // imprime um tab
  serial.print(analogValue, OCT);    // imprime um ASCII octal
  serial.print("\t");                 // imprime um tab
  serial.print(analogValue, BIN);    // imprime um ASCII binário
  serial.print("\t");                 // imprime um tab

  serial.print(analogValue/4, BYTE);
/* imprime como um byte único e adiciona um "cariage return"
 * (divide o valor por 4  pois analogRead() retorna número de 0 à 1023,
 * mas um byte pode armazenar valores somente entre 0 e 255
 */

  serial.print("\t");                 // imprime um tab

  delay(1000);        // espera 1 segundo para a próxima leitura
}
```

- **Serial.println(data)** Esta função envia dados para a porta serial seguidos por um *carriage return* (*ASCII* 13, ou '\r') e por um caractere de linha nova (*ASCII* 10, ou '\n'). Este comando utiliza os mesmos formatos do *Serial.print( )*:

  Serial.println(valor) imprime o valor de um número decimal em uma string *ASCII* seguido por um carriage return e um linefeed.

  Serial.println(valor, DEC) imprime o valor de um número decimal em uma string *ASCII* seguido por um carriage return e um linefeed.

  Serial.println(valor, HEX) imprime o valor de um número hexadecimal em uma string *ASCII* seguido por um carriage return e um linefeed.

  Serial.println(valor, OCT) imprime o valor de um número octal em uma string *ASCII* seguido por um carriage return e um linefeed.

  Serial.println(valor, BIN) imprime o valor de um número binário em uma string *ASCII* seguido por um carriage return e um linefeed.

  Serial.println(valor, BYTE) imprime o valor de um único byte seguido por um carriage return e um linefeed.

  Serial.println(str) se str for uma string ou um array de chars, imprime uma string *ASCII* seguido por um carriage return e um linefeed.

  Serial.println() imprime apenas um carriage return e um linefeed.

Exemplo:

```
/* Entrada Analógica
 lê uma entrada analógica no pino analógico 0 e imprime o valor na porta serial.
 */

int analogValue = 0;    // variável que armazena o valor analógico

void setup() {
  // abre a porta serial e justa a velocidade para 9600 bps:
  Serial.begin(9600);
}

void loop() {
  analogValue = analogRead(0);   // lê o valor analógico no pino 0:

  /* imprime em diversos formatos */
  Serial.println(analogValue);       // imprime um ASCII decimal - o mesmo que "DEC"
  Serial.println(analogValue, DEC);  // imprime um ASCII decimal
  Serial.println(analogValue, HEX);  // imprime um ASCII hexadecimal
  Serial.println(analogValue, OCT);  // imprime um ASCII octal
  Serial.println(analogValue, BIN);  // imprime um ASCII binário
  Serial.println(analogValue/4, BYTE); // imprime como um byte único
  delay(1000);// espera 1 segundo antes de fazer a próxima leitura:
}
```

4 Exemplo de Aplicação 1: Imprimindo uma mensagem no LCD

Componentes: 1 LCD, 1 potenciômetro

Neste exemplo mostraremos como conectar corretamente um LCD ao Arduino, além de imprimir o famoso "Hello World!" na tela do LCD através da função lcd.print(), contida na biblioteca **LiquidCrystal.h**.

Sugestão de montagem

Para conectar a tela LCD ao Arduino, conecte os seguintes pinos:

- pino VSS(1) do LCD ao pino GND
- pino VDD(2) do LCD ao pino 5V
- pino RS(4) do LCD ao pino 12
- pino RW(5) do LCD ao pino GND
- pino Enable(6) do LCD ao pino 11
- pino D4(11) do LCD ao pino 5
- pino D5(12) do LCD ao pino 4
- pino D6(13) do LCD ao pino 3
- pino D7(14) do LCD ao pino 2

Devemos conectar também o potenciômetro de 10K aos pinos 5V, GND e V0(3) do LCD, conforme sugere a Figura 4:

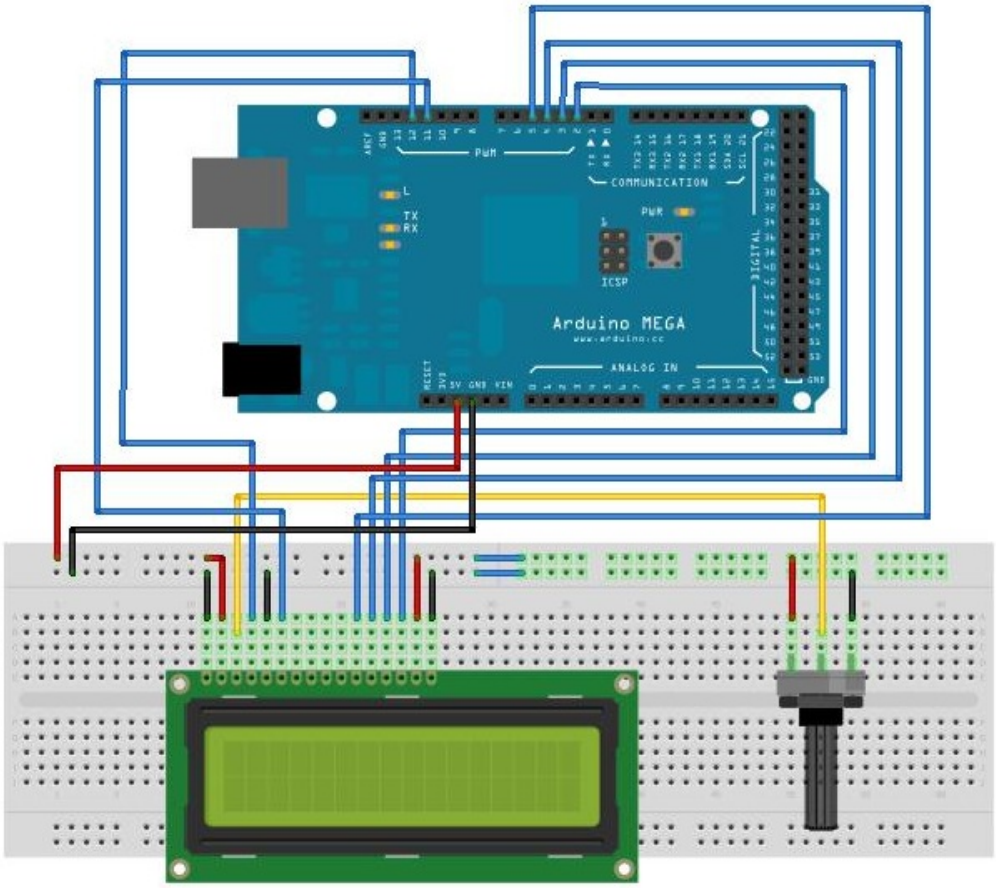

Figura 4: Montagem do Circuito

Código fonte

```
#include <LiquidCrystal.h>
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);
void setup() {
   lcd.begin(16, 2);
   lcd.print("Hello World!");
}
void loop() {
  lcd.setCursor(0, 1);
  lcd.print(millis()/1000);
  lcd.print("s");
}
```

5 Exemplo de Aplicação 2: Alterando a frequência com que o LED pisca

Componentes: 1 Potênciometro, 1 LED

Este projeto é muito simples e tratará da utilização do potenciômetro, que é um componente eletrônico que possui resistência elétrica ajustável. A frequência com que o LED pisca vai depender diretamente do ajuste do potenciômetro.

Sugestão de montagem

Conecte um potenciômetro na porta 0 e um LED na porta 11, com uma resistência de 220 Ohms como mostra a Figura 5.

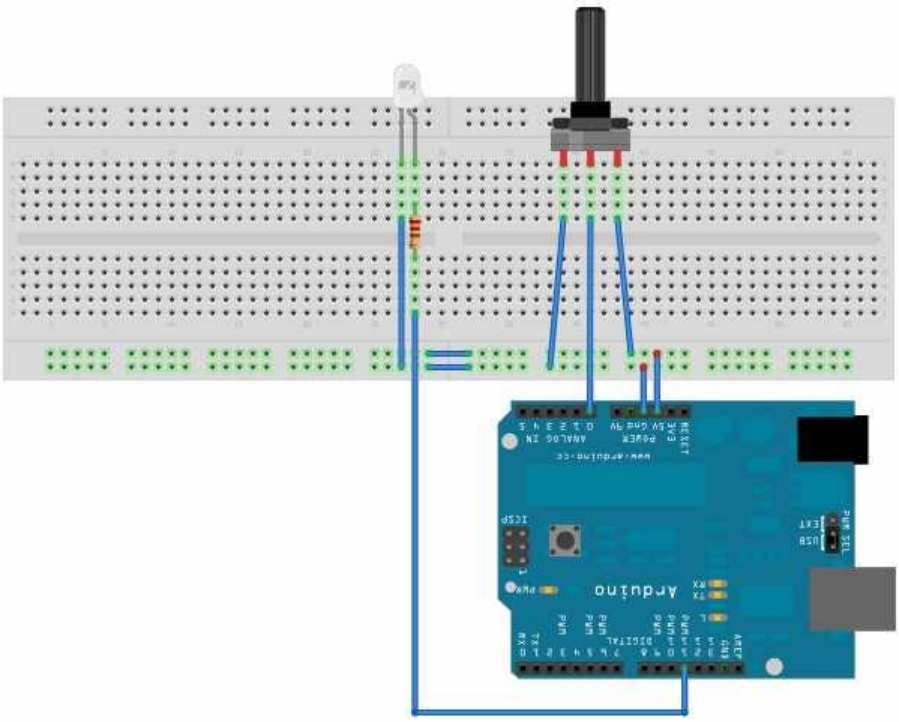

Figura 5: Montagem do Circuito

Código-fonte

```
int potPin = 0;
int ledPin = 11;
int val = 0;
void setup() {
  pinMode(ledPin, OUTPUT);
}

void loop() {
  val = analogRead(potPin);
  digitalWrite(ledPin, HIGH);
  delay(val);
  digitalWrite(ledPin, LOW);
  delay(val);
}
```

6 Exemplo de Aplicação 3: Semáforo de Carros e Pedestres

Componentes: 2 LEDs vermelho, 2 LEDs verdes, 1 LED amarelo

Neste exemplo, vamos simular o trânsito em uma determinada rua de Campo Grande. Para isso, você deseja controlar com segurança e eficiência, o fluxo de carros e de pedestres. Elabore um projeto para implantação de dois semáforos nessa rua: um que controle a circulação de carros e outra que garanta a segurança dos pedestres para atravessar a rua, como mostra a Figura 6, obedecendo as seguintes regras:

- quando o sinal do semáforo de carro estiver com as cores verde ou amarelo aceso, o sinal vermelho de pedestres deve estar aceso.
- quando o sinal vermelho do semáforo de carro estiver aceso, somente o sinal verde de pedestres deve ficar aceso.
- caso o botão seja apertado, a preferência de passagem pela rua é do pedestre.

Sugestão de montagem

Para o semáforo de carros: conecte um pino verde na porta 22, um pino amarelo na porta 23 e um pino vermelho na porta 24; Para o semáforo de pedestres: conecte um pino verde na porta 28, um pino vermelho na porta 29 e um botão na porta 2, como mostra a Figura 6. Dica: em seu programa, todos os LEDs devem estar configurado como saída e o botão deve estar configurado como entrada.

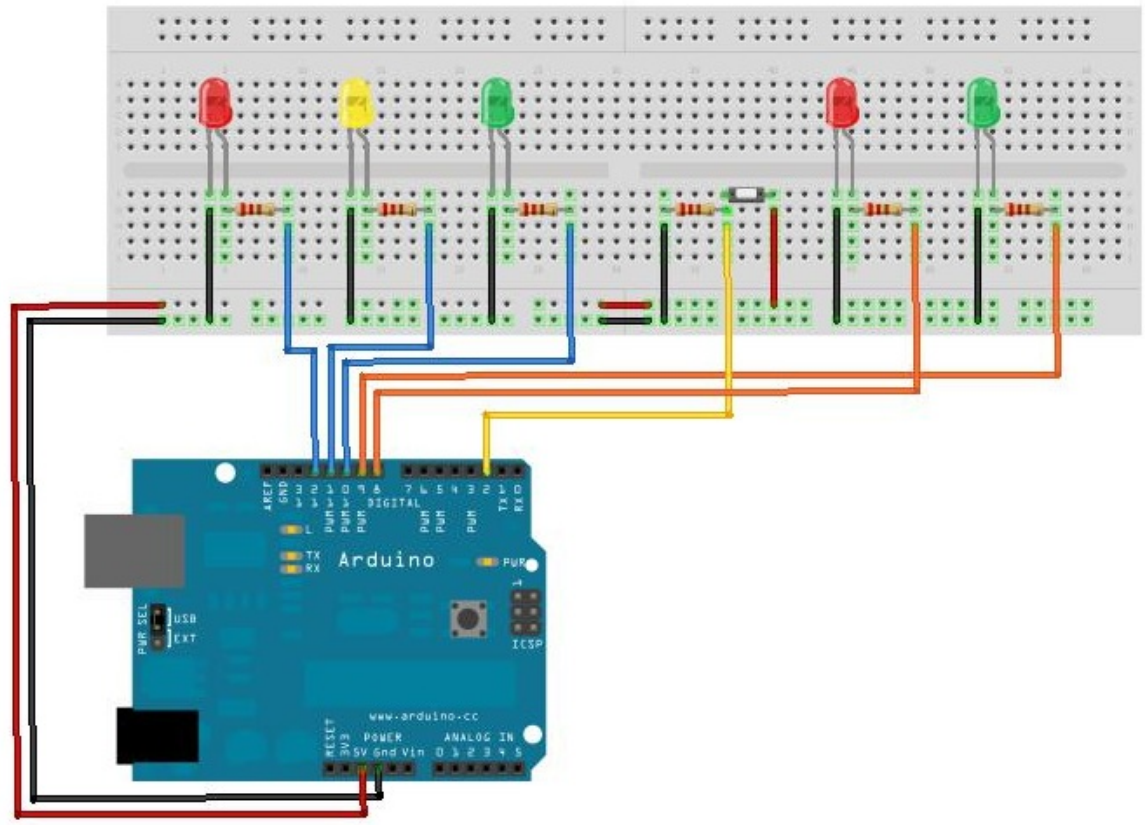

Figura 6: Montagem do Circuito

Código-Fonte

```
const int scVerde = 10;
const int scAmarelo = 11;
const int scVermelho = 12;
const int spVerde = 8;
const int spVermelho = 9;
int ledState = LOW;
long previousMillis = 0;
long interval = 5000;
int ctrlLuz = 0;

void setup() {
```

```
  Serial.begin(9600);
  pinMode(scVerde,OUTPUT);
  pinMode(scAmarelo,OUTPUT);
  pinMode(scVermelho,OUTPUT);
  pinMode(spVerde,OUTPUT);
  pinMode(spVermelho,OUTPUT);
  pinMode(2, INPUT); // Botao
}

void loop() {
  unsigned long currentMillis = millis();
  int sensorValue = digitalRead(2);
  if(currentMillis - previousMillis > interval) {
    previousMillis = currentMillis;

    switch(ctrlLuz) {
      case 0 : // Verde
               digitalWrite(scVermelho,LOW);
               digitalWrite(scVerde,HIGH);
               digitalWrite(spVerde,LOW);
               digitalWrite(spVermelho,HIGH);
               ctrlLuz++;
               interval = 15000;
               break;
      case 1 : // amarelo
               digitalWrite(scVerde,LOW);
               digitalWrite(scAmarelo,HIGH);
               digitalWrite(spVerde,LOW);
               digitalWrite(spVermelho,HIGH);
               ctrlLuz++;
               interval = 1000;
               break;
      case 2 : // Vermelho
               digitalWrite(scAmarelo,LOW);
               digitalWrite(scVermelho,HIGH);
               digitalWrite(spVermelho,LOW);
               digitalWrite(spVerde,HIGH);
               interval = 7000;
               ctrlLuz = 0;
               break;
```

```
    }
  }
  if((sensorValue == 1) && (ctrlLuz == 1)) {
    interval = 2000;
    Serial.print("Sensor ");
    Serial.println(sensorValue, DEC);
  }
}
```

7 Exemplo de Aplicação 4: Termômetro

Componentes: 1 Sensor de Temperatura, 2 LEDs vermelho, 2 LEDs amarelo, 2 LEDs verde, 1 Buzzer

É possível simularmos um termômetro utilizando o Kit Arduino, utilizando LEDs e um sensor de temperatura. Dependendo do valor que o sensor ler a respeito da temperatura ambiente, ele acende n LEDs que correspondente a temperatura lida. Para ilustrar melhor, imagine um circuito com 20 LEDs onde cada LED correspondesse a 1^oC. Caso o sensor ler uma temperatura de 15^oC em uma sala, isso significa que os 15 primeiros LEDs desse circuito deverão acender. Como essa escala não é possível para nós, implemente um termômetro que utilize 6 LEDs na qual cada um representa uma determinada unidade de temperatura em escala. Para incrementar nosso projeto, faça com que quando o termômetro indicar uma situação crítica de temperatura no ambiente, ou seja, quando todos os LEDs estiverem acessos, um Buzzer é acionado, indicando uma alta temperatura ambiente. Seu projeto deve seguir o seguinte padrão:

Se o Sensor de temperatura ler um valor

- maior que 30: ligue o primeiro LED verde, contando da esquerda. Caso contrário, mantenha-o apagado.
- maior que 35: ligue o segundo LED verde. Caso contrário, mantenha-o apagado.
- maior que 40: ligue o primeiro LED amarelo. Caso contrário, mantenha-o apagado.
- maior que 45: ligue o segundo LED amarelo. Caso contrário, mantenha-o apagado.
- maior que 50: ligue o primeiro LED vermelho. Caso contrário, mantenha-o apagado.
- maior que 55: ligue o segundo LED vermelho. Caso contrário, mantenha-o apagado.

Lembre-se que, caso todos os LEDs estiverem ativos, isso significa que o termômetro detectou uma temperatura crítica no ambiente e um alarme deve ser soado.

Sugestão de montagem

Conecte um LED verde na porta 8 e outro na porta 9; um LED amarelo na porta 10 e outro na porta 11; e por fim, um LED vermelho na porta 12 e um outro na porta 13. Conecte na porta 6, o *Buzzer* e o Sensor de temperatura na porta 0. Observe a Figura 7.

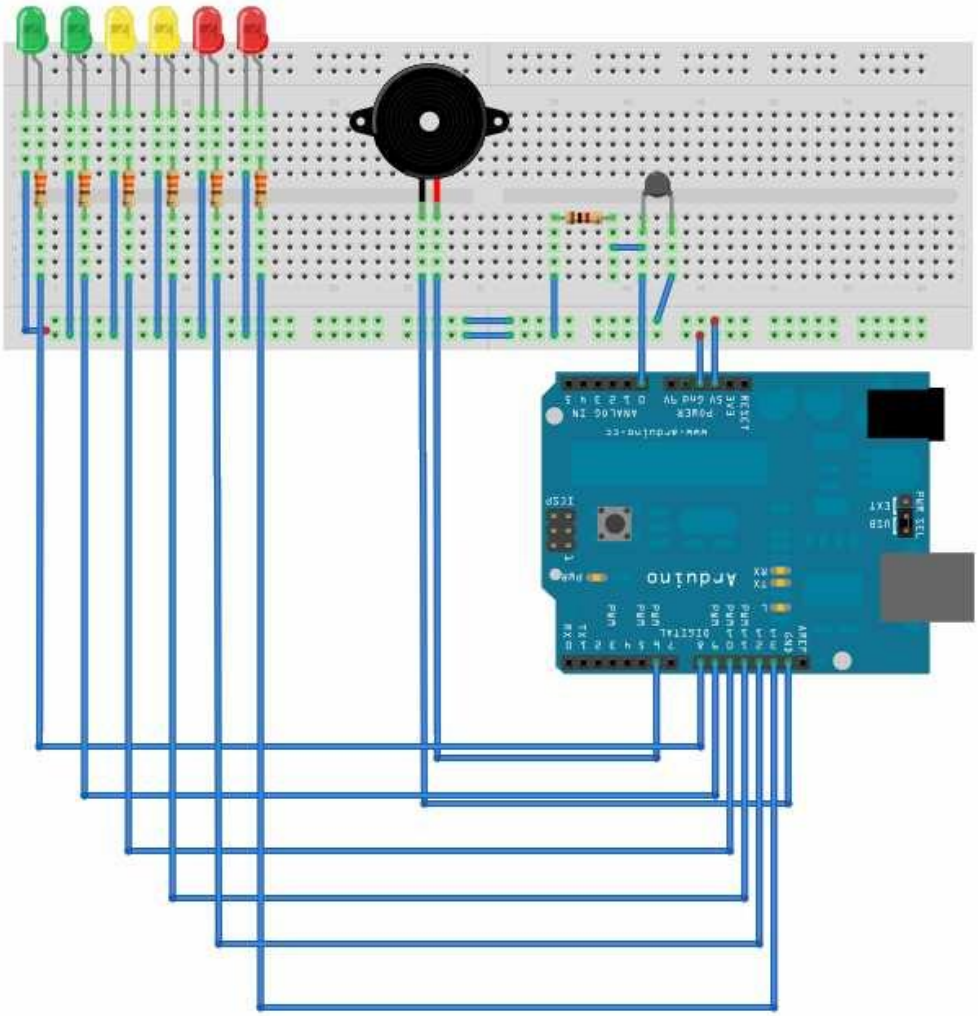

Figura 7: Montagem do Circuito

Código-Fonte

```
int PinoSensor = 0;
int Buzzer = 6;
int led1 = 8;
int led2 = 9;
int led3 = 10;
int led4 = 11;
int led5 = 12;
int led6 = 13;
int ValorSensor = 0;

void setup() {
  pinMode(Buzzer, OUTPUT);
  pinMode(led1, OUTPUT);
  pinMode(led2, OUTPUT);
  pinMode(led3, OUTPUT);
  pinMode(led4, OUTPUT);
  pinMode(led5, OUTPUT);
  pinMode(led6, OUTPUT);
  Serial.begin(9600);
}

void loop(){
  ValorSensor = analogRead(PinoSensor);
  Serial.print("Valor do Sensor = ");
  Serial.println(ValorSensor);
  if (ValorSensor > 30)
    digitalWrite(led1, HIGH);
  else
    digitalWrite(led1, LOW);
  if (ValorSensor > 35)
    digitalWrite(led2, HIGH);
  else
    digitalWrite(led2, LOW);
  if (ValorSensor > 40)
    digitalWrite(led3, HIGH);
  else
    digitalWrite(led3, LOW);
  if (ValorSensor > 45)
```

```
    digitalWrite(led4, HIGH);
  else
    digitalWrite(led4, LOW);
  if (ValorSensor > 50)
    digitalWrite(led5, HIGH);
  else
    digitalWrite(led5, LOW);
  if (ValorSensor > 55 ){
    digitalWrite(led6, HIGH);
    digitalWrite(Buzzer, HIGH);
  }
  else{
    digitalWrite(led6, LOW);
    digitalWrite(Buzzer, LOW);
  }
  delay(1000);
}
```

8 Exemplo de Aplicação 5: Piano

Componentes: 3 Botões, 3 Leds, 1 Buzzer

É possível "fazer barulho", ou até mesmo tocar notas musicais com o kit arduino, através de um componente chamado buzzer. O buzzer funciona como um pequeno alto-falante que, quando alimentado por uma fonte, componentes metálicos internos vibram na frequência da fonte, produzindo assim um som. O buzzer não tem capacidade o suficiente para tocar músicas, mas consegue produzir apitos, úteis em sirenes e alarmes por exemplo.
Implemente seu projeto de forma que quando pressionado um botão, toque uma nota musical e acenda um LED. Como temos apenas 3 botões e sete notas musicais, cada botão vai referenciar a mais de uma nota musical, assim como os LEDs.

Obs: As notas musicais são: dó, ré, mi, fa, sol, la, si.

Sugestão de montagem

Conecte cada um dos botões nas portas 2, 3 e 4. Conecte o *Buzzer* na porta 10 e cada um dos LEDs, nas portas 11, 12 e 13. Observe a Figura 8.

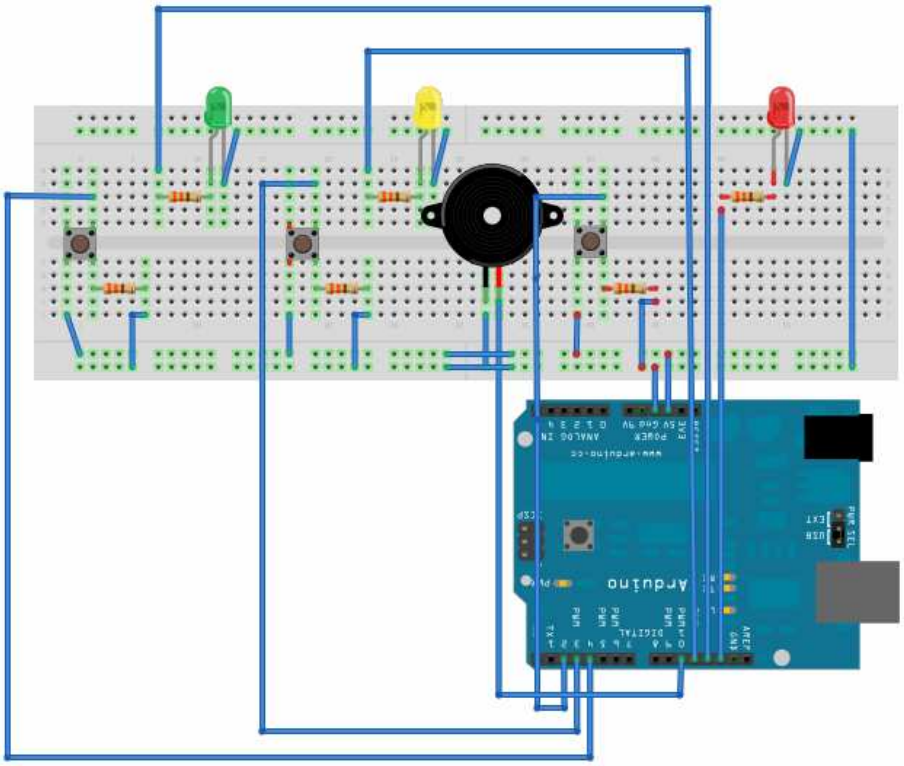

Figura 8: Montagem do Circuito

Código-fonte

```
const int ledPin1 = 13;
const int ledPin2 = 12;
const int ledPin3 = 11;
const int Botao1 = 2;
const int Botao2 = 3;
const int Botao3 = 4;
const int Buzzer = 10;
int EstadoBotao1 = 0;
int EstadoBotao2 = 0;
int EstadoBotao3 = 0;
int Tom = 0;

void setup() {
  pinMode(Buzzer, OUTPUT);
  pinMode(ledPin1, OUTPUT);
  pinMode(Botao1, INPUT);
  pinMode(ledPin2, OUTPUT);
  pinMode(Botao2, INPUT);
  pinMode(ledPin3, OUTPUT);
  pinMode(Botao3, INPUT);
}

void loop(){
  EstadoBotao1 = digitalRead(Botao1);
  EstadoBotao2 = digitalRead(Botao2);
  EstadoBotao3 = digitalRead(Botao3);
  if(EstadoBotao1 && !EstadoBotao2 && !EstadoBotao3) {
    Tom = 50;
    digitalWrite(ledPin1, HIGH);
  }
  if(EstadoBotao2 && !EstadoBotao1 && !EstadoBotao3) {
    Tom = 400;
    digitalWrite(ledPin3, HIGH);
  }
  if(EstadoBotao3 && !EstadoBotao2 && !EstadoBotao1) {
    Tom = 1000;
    digitalWrite(ledPin2, HIGH);
```

```
  }
  while(Tom > 0){
    digitalWrite(Buzzer, HIGH);
    delayMicroseconds(Tom);
    digitalWrite(Buzzer, LOW);
    delayMicroseconds(Tom);
    Tom = 0;
    digitalWrite(ledPin1, LOW);
    digitalWrite(ledPin2, LOW);
    digitalWrite(ledPin3, LOW);
  }
}
```

9 Exemplo de Aplicação 6: Alarme

Componentes: 1 Sensor de Distâcia, 1 Buzzer, 1 Led

Hoje em dia é comum encontrarmos sensores de distância instalados na traseira de carros para auxiliarem os motoristas na hora de fazerem balizas. Esses sensores detectam objetos que estão em uma determinada distância e caso o sensor decide que o objeto está muito perto do carro, ele emite um *beep*. Os sensores de distância tem várias aplicações no meio comercial e industrial, como por exemplo, também são utilizados em residências e em escritórios para indicar a presença de alguém no ambiente. Alguns desses sensores emitem um *beep* e outros são mais discretos: acendem uma luz. Implemente um projeto onde o sensor ultra-sonico acenda um LED ou emita um *beep* quando um objeto estiver a menos de 30 cm do seu raio de alcance.

Sugestão de montagem

Configuere o pino 13 do kit Arduino para a conexão do LED. Reserve o pino 7 para o sensor de distância. Conecte o *Buzzer* no pino 10.

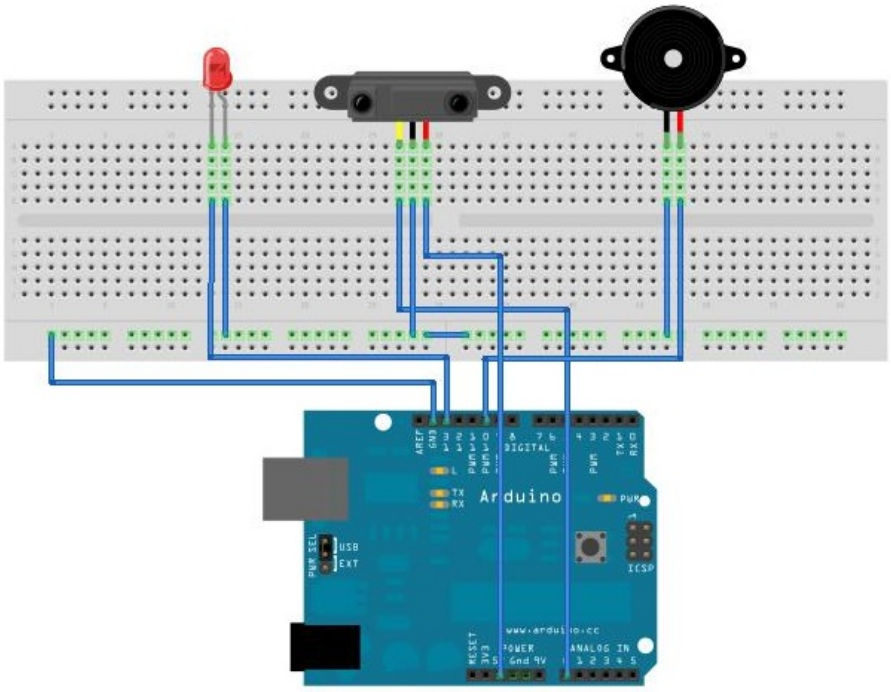

Figura 9: Montagem do Circuito

Código-fonte

```
int LED = 13;
int buzzer = 10;
int sharp = 0;

void setup() {
  pinMode(sharp,INPUT);
  pinMode(buzzer,OUTPUT);
  pinMode(LED,OUTPUT);
}

void loop() {
  int ir = analogRead(sharp);
  if(ir>150) {
    digitalWrite(LED, HIGH);
    digitalWrite(buzzer, HIGH);
  }
  else {
    digitalWrite(LED, LOW);
    digitalWrite(buzzer, LOW);
  }
}
```

10 Exemplo de Aplicação 7: Projeto Alarme Multipropósito

Componentes: 2 Leds Verdes, 2 Leds Amarelos, 2 Leds Vermelhos, 1 Sensor de Luminosidade LDR, 1 Sensor de Temperatura NTC, 1 Led Alto Brilho, 1 Buzzer

Neste exemplo temos que ter um cuidado maior, por ele ser mais elaborado que os anteriores. Adotamos neste projeto que 3 LEDs (1 de cada cor) correspondem à temperatura, os outros 3 correspondem à luminosidade. Você deve implementar seu projeto da seguinte forma:

- A medida que a temperatura for aumentando vai acendendo os LEDs correspondentes à ela, um por um, então se a temperatura estiver alta os 3 LEDs devem estar acesos e um alarme deve soar.
- Se a luminosidade do ambiente estiver alta os 3 LEDs correspondentes à ela devem estar acesos, a medida que a luminosidade for ficando fraca, ou seja, o ambiente for ficando escuro, os LEDs vão apagando um por um, até que todos os LEDs estejam apagados indicando falta total de luminosidade no ambiente, nesse momento o LED de alto brilho deve acender.
- Se os 3 LEDs de temperatura estiverem acesos e os 3 LEDs de luminosidade apagados, então deve soar o sinal e o LED de alto brilho deve acender.

Sugestão de montagem

Conecte os LEDs nos pinos de 5 à 11. Conecte o *Buzzer* na porta 2, o sensor de temperatura na porta 1, e o sensor de luminosidade na porta 0.

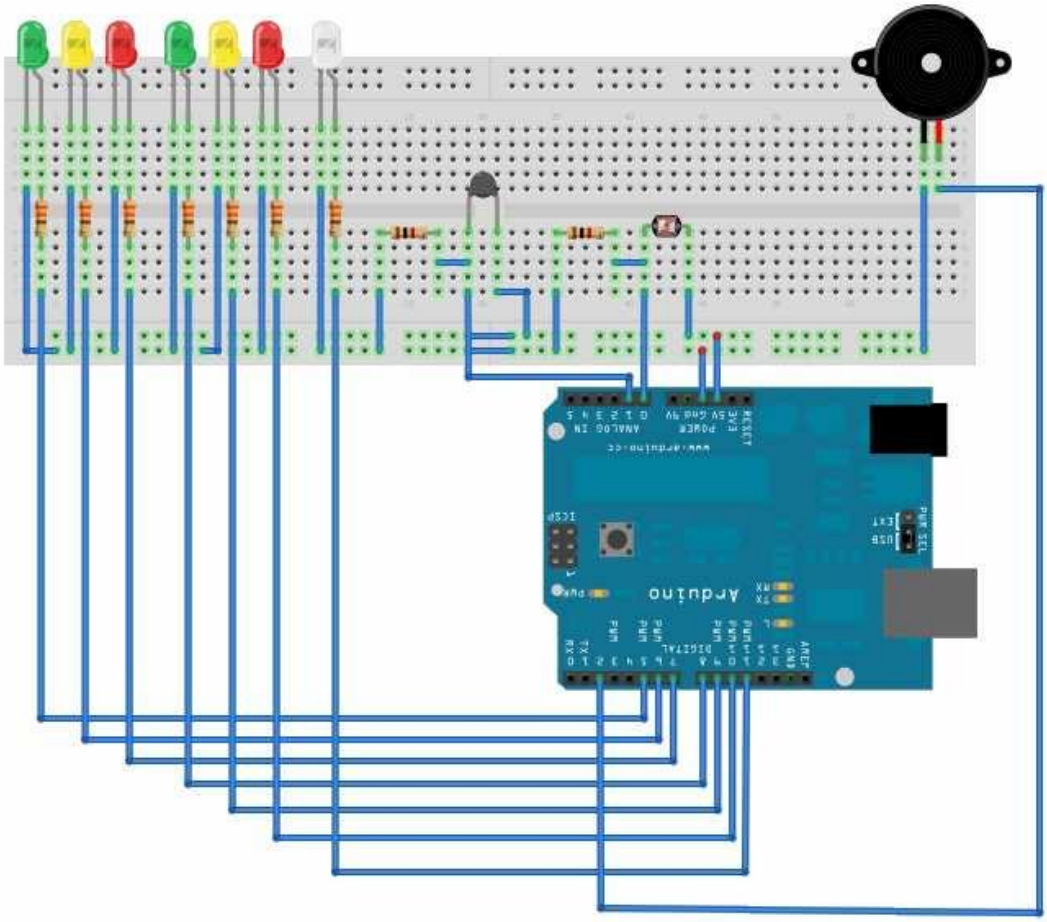

Figura 10: Montagem do Circuito

Código-fonte

```
const int LDR = 0;
const int NTC = 1;
const int Buzzer = 2;
const int led1 = 5;
const int led2 = 6;
const int led3 = 7;
const int led4 = 8;
const int led5 = 9;
const int led6 = 10;
const int ledAB = 11;
int ValorLDR = 0;
int ValorNTC = 0;

void setup(){
  pinMode(Buzzer, OUTPUT);
  pinMode(led1, OUTPUT);
  pinMode(led2, OUTPUT);
  pinMode(led3, OUTPUT);
  pinMode(led4, OUTPUT);
  pinMode(led5, OUTPUT);
  pinMode(led6, OUTPUT);
  pinMode(ledAB, OUTPUT);
  Serial.begin(9600);
}

void loop(){
  ValorLDR = analogRead(LDR);
  ValorNTC = analogRead(NTC);
  Serial.print("Valor da Temperatura = ");
  Serial.println(ValorNTC);
  if (ValorNTC > 10)
    digitalWrite(led1, HIGH);
  else
    digitalWrite(led1, LOW);

  if (ValorNTC > 20)
    digitalWrite(led2, HIGH);
  else
```

```
    digitalWrite(led2, LOW);

  if (ValorNTC > 30){
    digitalWrite(led3, HIGH);
    digitalWrite(Buzzer, HIGH);
  }
  else{
    digitalWrite(led3, LOW);
    digitalWrite(Buzzer, LOW);
  }
  if (ValorLDR > 600)
    digitalWrite(led6, HIGH);
  else
    digitalWrite(led6, LOW);

  if (ValorLDR > 500)
    digitalWrite(led5, HIGH);
  else
    digitalWrite(led5, LOW);

  if (ValorLDR > 450){
    digitalWrite(led4, HIGH);
    digitalWrite(ledAB, LOW);
  }
  else{
    digitalWrite(led4, LOW);
    digitalWrite(ledAB, HIGH);
  }
}
```

11 Exemplo de Aplicação 8: Portão Eletrônico

Componentes: 2 servo-motores, 1 sensor de distância, barra retangular de cartolina ou papelão.

Neste exemplo utilizaremos servo-motores para realizar a elevação de uma barra retangular simulando o funcionamento de um portão eletrônico. Além dos servo-motores, utilizaremos também um sensor de distância para acionar os sensores caso a distância seja menor que determinado limite em centímetros. O funcionamento do circuito acontece da seguinte forma:

- O sensor de distância deve ser posicionado a frente (utiliza uma segunda *protoboard*) dos servo-motores.
- Se a distância retornada pelo sensor for menor ou igual a determinado limite (10cm), os servo-motores devem ser acionados para girar 90^o para esquerda (sentido anti-horário).
- Uma vez que a distância retornada pelo sensor seja maior que 10cm (indica que o objeto está saindo do campo de detecção do sensor), o sensor aguarda 3 segundos e aciona os servo-motores para retornarem a posição inicial (giro de 90^o para direita).

Sugestão de montagem

Siga o exemplo da Figura 11

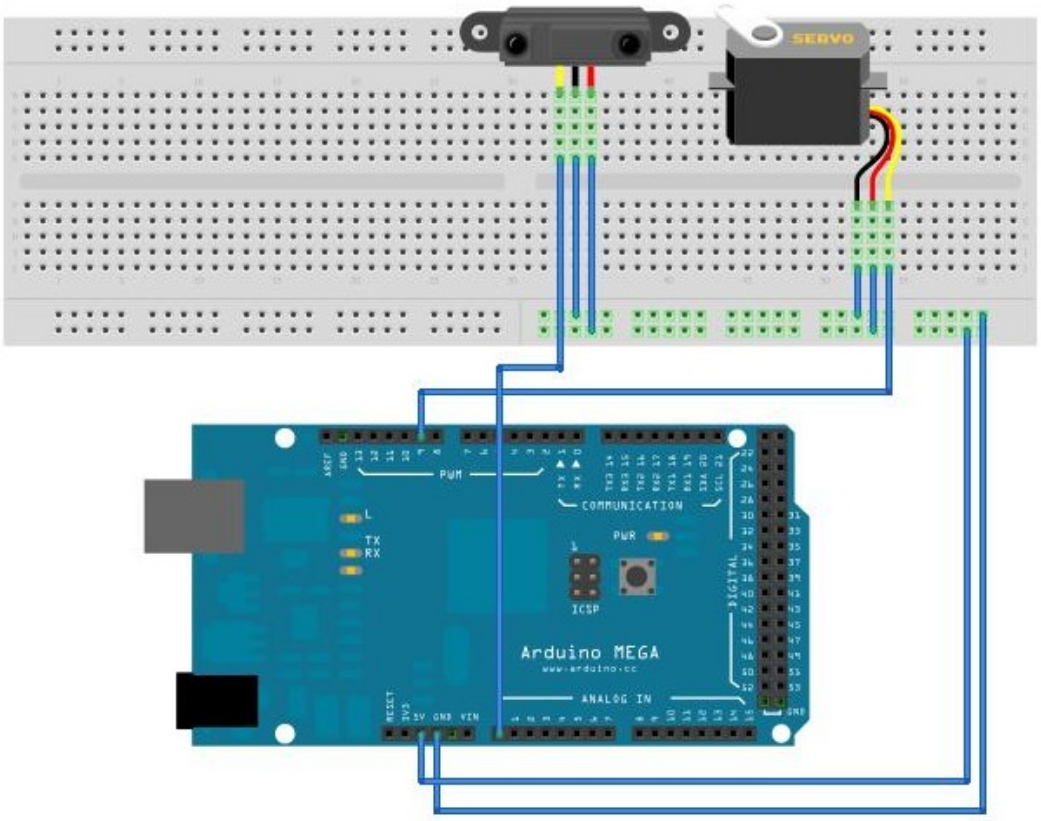

Figura 11: Montagem do Circuito

Código-fonte

```
#include <Servo.h>
Servo myservo;

const int sensor = 0;

void setup() {
  myservo.attach(9);
  pinMode(sensor, INPUT);
  Serial.begin(9600);
}
```

```
void loop()
{
   int ir = analogRead(sensor);
   Serial.print(ir);
   Serial.print(" ir");
   Serial.println();
   if(ir > 450){
     myservo.write(90);
     delay (6000);
     myservo.write(-90);
   }
}
```