Instituto Federal de Educação, Ciência e Tecnologia de Mato Grosso do Sul

GREAT - Group of Robotics and EducAtional Technologies

Introdução ao Arduino

Conceitos Gerais e Programação

Equipe:

Prof. Leandro de Jesus

Prof. Marcia Ferreira Cristaldo

Sumário

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* [ver apêndice C.18] baseado em uma placa de circuito impresso [ver apêndice C.6] dotada de vários recursos de interfaceamento (pinagem de entrada e saída) e um microcontrolador Atmel AVR. É um projeto descendente da plataforma *Wiring* [ver apêndice C.25] 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*[ver apêndice C.19] .

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 enten-

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

dida por outros usuários.

- 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) [ver apêndice C.9], dos quais 14 podem ser usadas como saídas PWM [ver apêndice C.20] de 8 bits, 16 entradas analógicas, 4 UARTs [ver apêndice C.24] 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*) [ver apêndice C.13], 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.

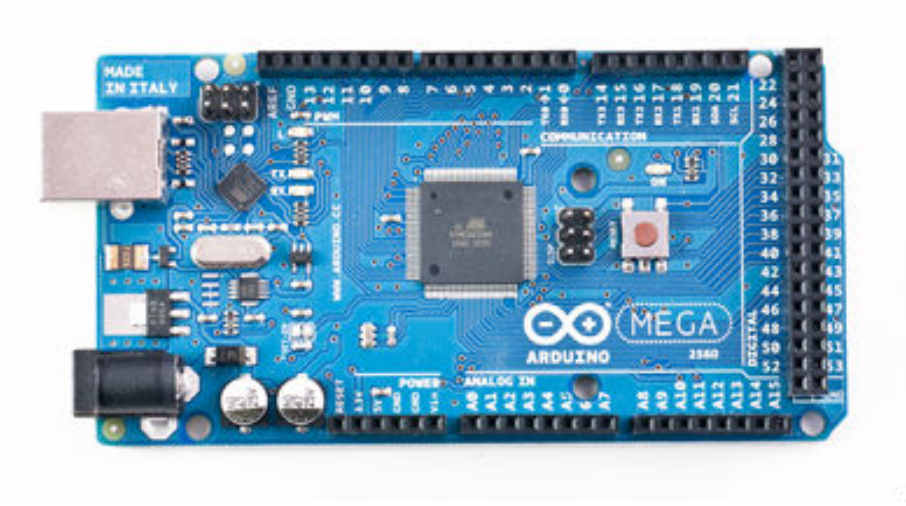

Figura 1: Arduino Mega 2560

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 [ver apêndice C.7]	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 oposicão 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 [ver apêndice C.10]. A corrente máxima é de 50 mA.
- **GND**. Pinos de referência (0V).

2.2 Memória

O ATmega2560 tem 256 kB de memória flash [ver apêndice C.16] para armazenamento de código (dos quais 8 kB é usado para o *bootloader* [ver apêndice C.3], 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** [ver apêndice C.22] por meio da *biblioteca SPI*. [ver apêndice C.2] 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 [ver apêndice C.23] 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) [ver apêndice C.21] .

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 [ver apêndice C.11] 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* [ver apêndice C.14] 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

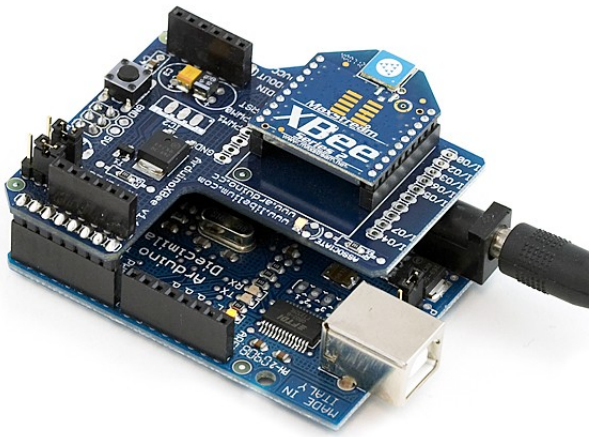

Figura 2: Exemplo de utilização do Arduino com kit acessório Xbee.

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*.

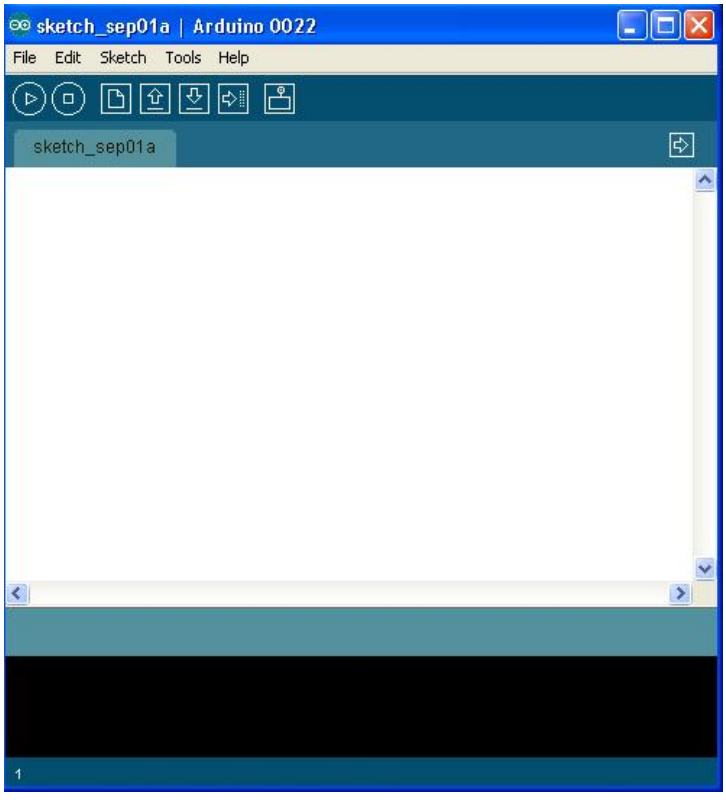

Figura 3: Ambiente de desenvolvimento (IDE) do Arduino

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* [ver apêndice C.5]. 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 [ver apêndice C.1]. 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** [ver apêndice C.8].

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 [ver apêndice C.12]. 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		

Estrutura

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 7, 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) [ver apêndice C.17]. 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. [ver apêndice C.4]

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 será mostrado 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 o 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

Deve-se conectar também o potenciômetro de 10K Ohms aos pinos 5V, GND e V0(3) do LCD, conforme sugere as Figuras 4 e 5:

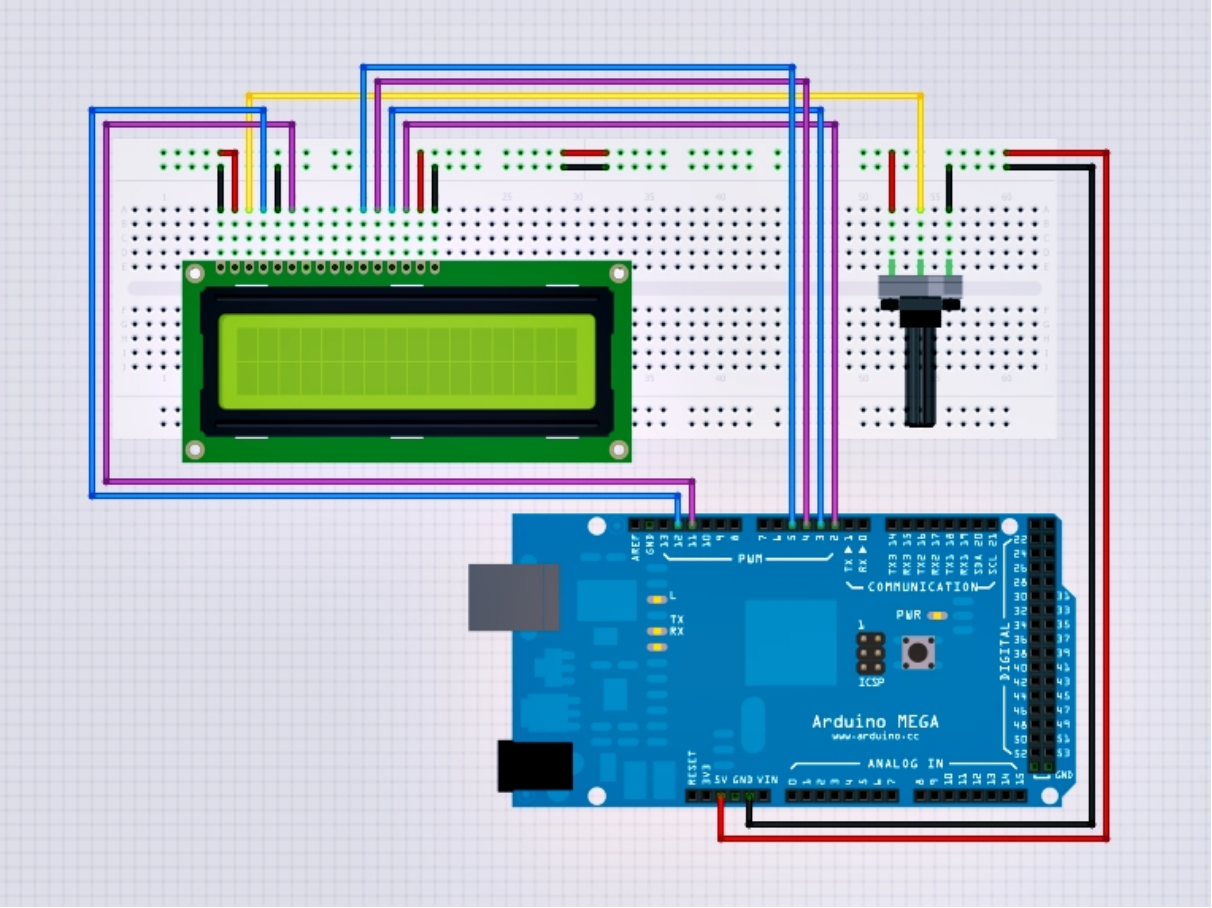

Figura 4: Montagem do Circuito

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");
}
```

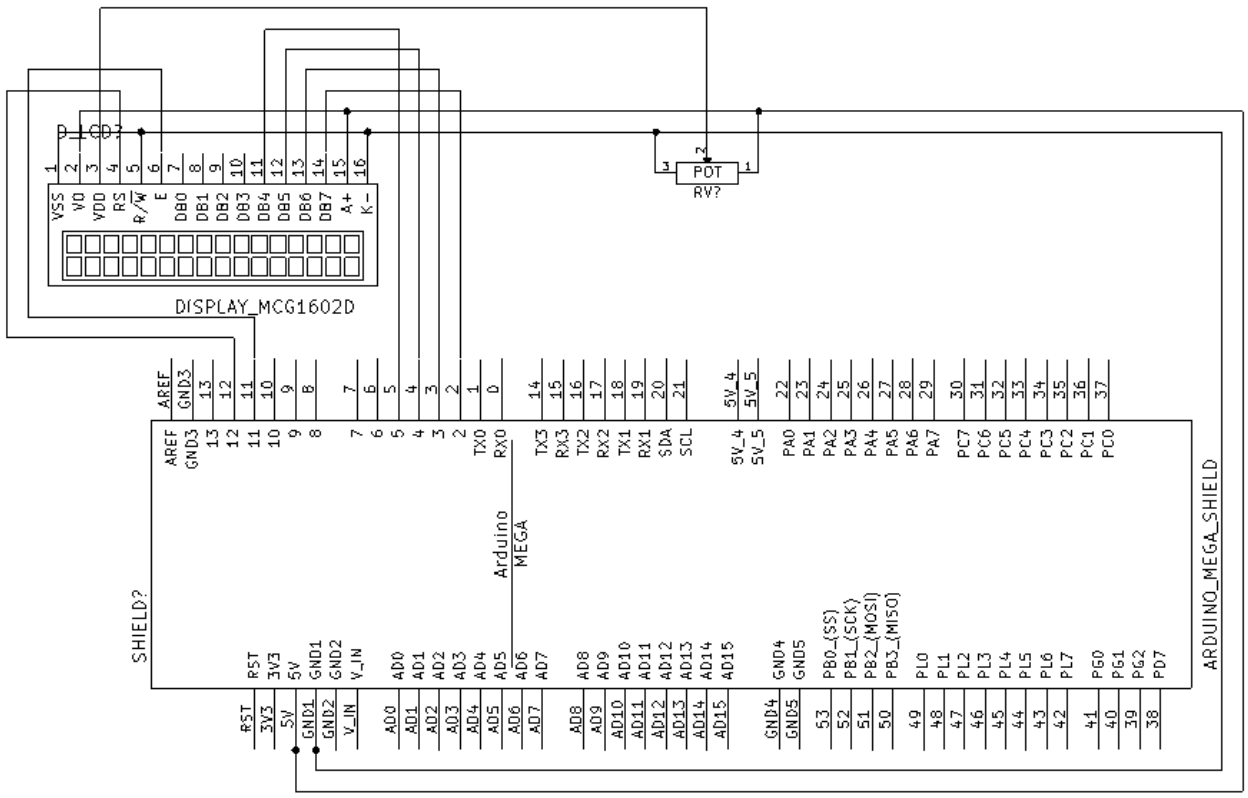

Figura 5: Esquemático Eletrônico

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 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 um resistor de 220 Ohms, como mostra as Figuras 6 e 7.

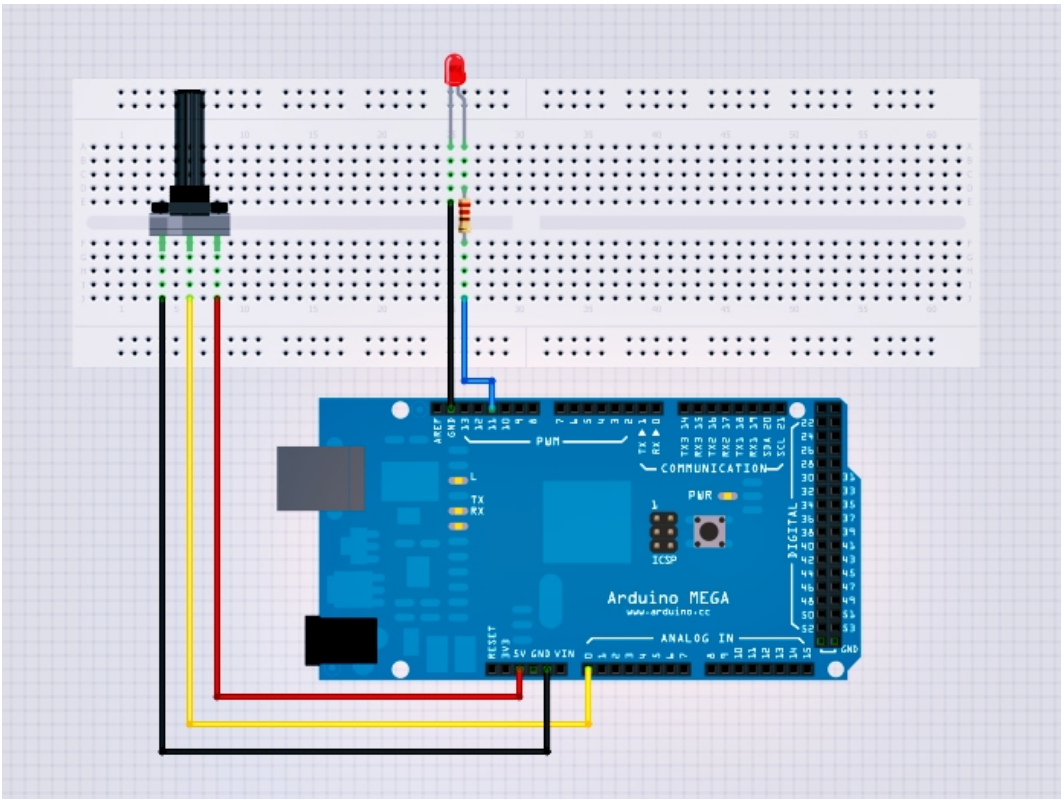

Figura 6: Montagem do Circuito

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);
}
```

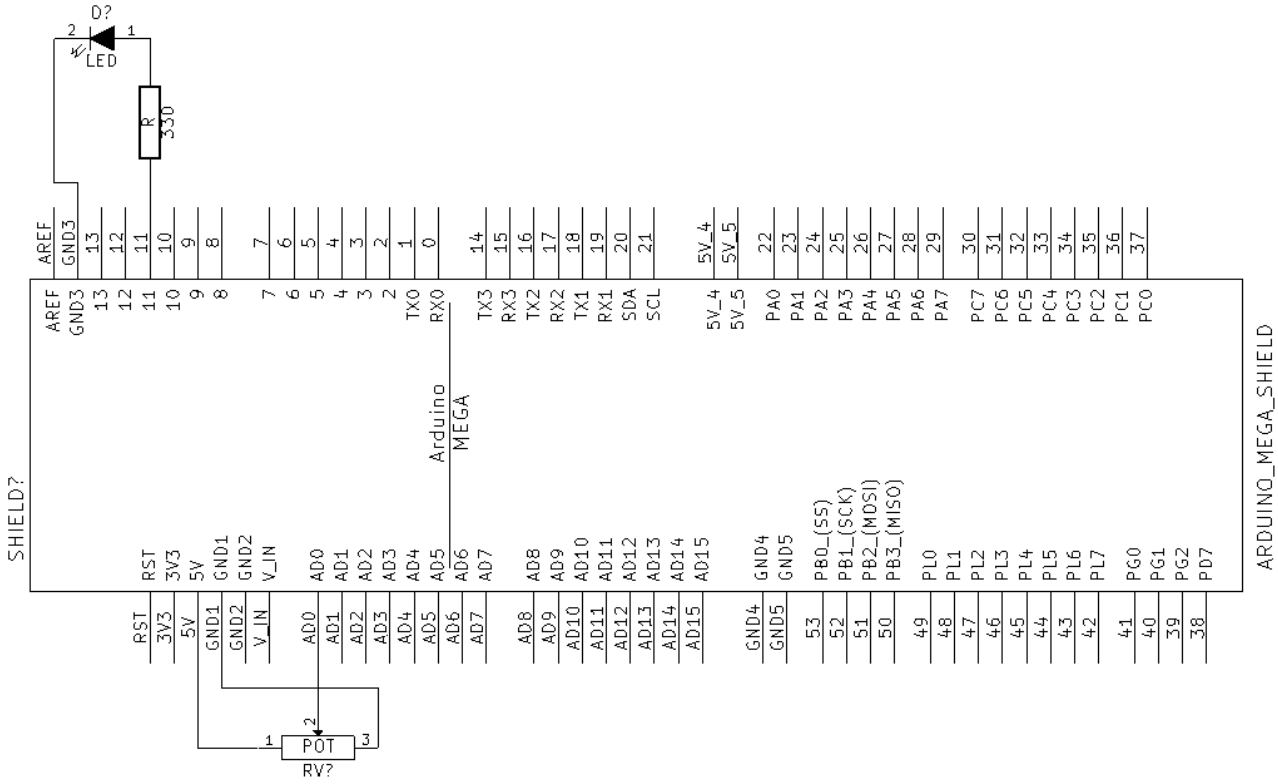

Figura 7: Esquemático Eletrônico

6 Exemplo de Aplicação 3: Semáforo de Carros e Pedestres

Componentes: 2 LEDs vermelho, 2 LEDs verdes, 1 LED amarelo, 1 *push-button*

Neste exemplo, será simulado o trânsito em uma determinada rua de Campo Grande, e deseja-se 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 outro que garanta a segurança dos pedestres para atravessar a rua, como mostra a Figura 8, obedecendo as seguintes regras:

- quando o sinal do semáforo de carros estiver com as cores verde ou amarelo acesas, o sinal vermelho de pedestres deve estar aceso.
- quando o sinal vermelho do semáforo de carros 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 LED verde na porta 10, um LED amarelo na porta 11 e um LED vermelho na porta 12; Para o semáforo de pedestres: conecte um LED verde na porta 8, um LED vermelho na porta 9 e um *push-button* na porta 2, como mostra as Figuras 8 e 9. Dica: em seu programa, todos os LEDs devem estar configurados como saída e o botão deve estar configurado como entrada.

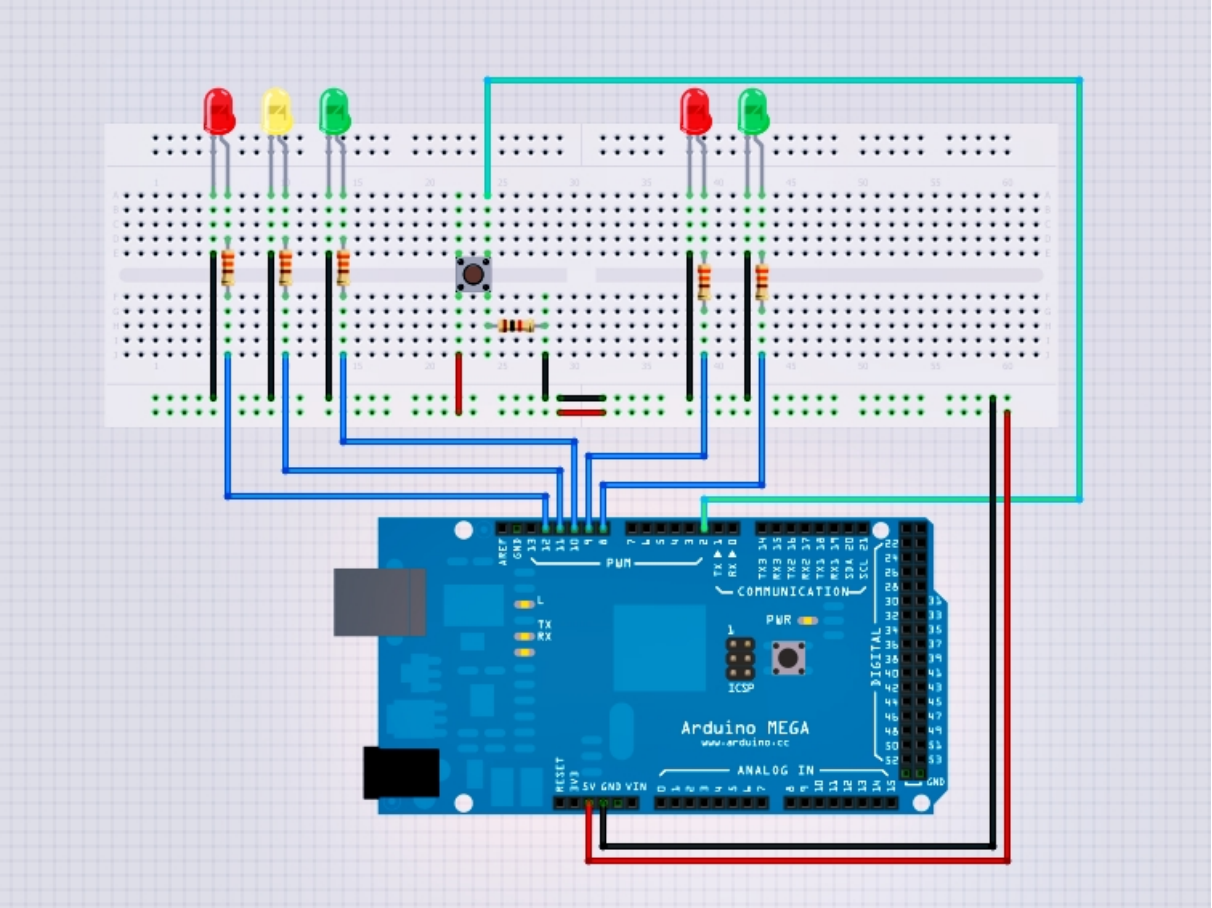

Figura 8: Montagem do Circuito

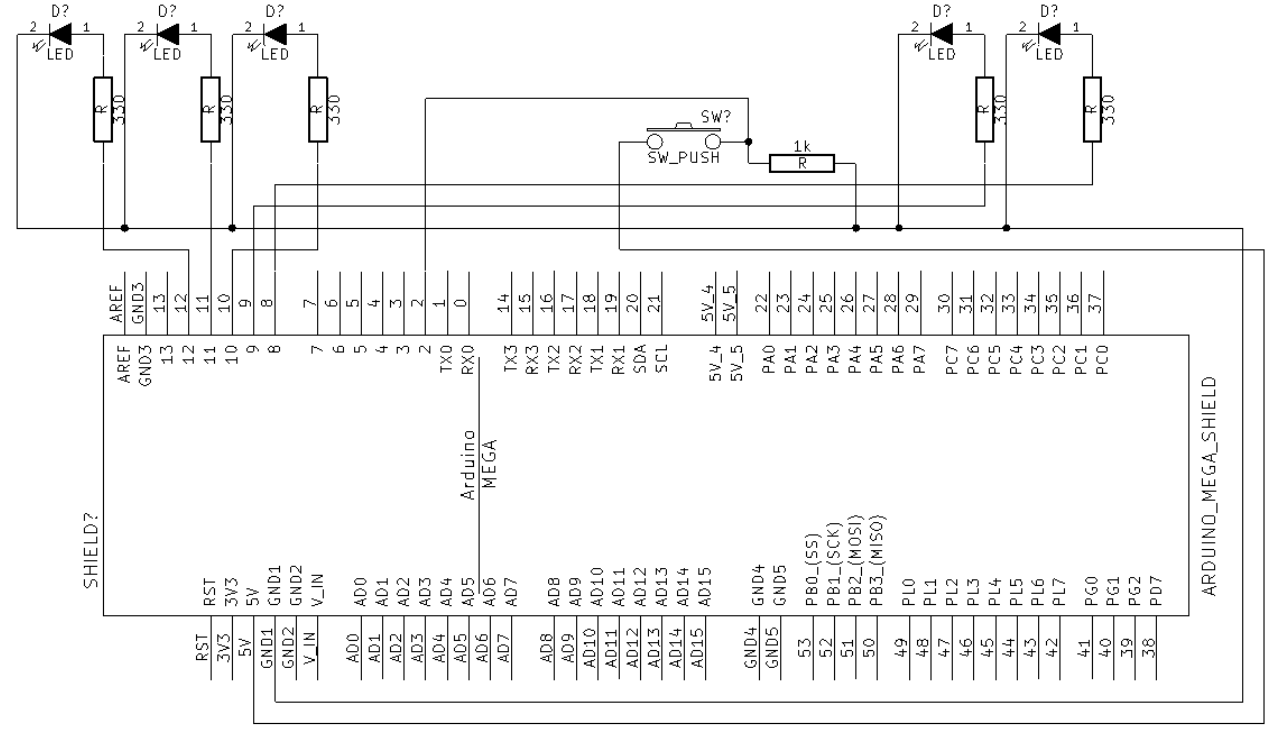

Figura 9: Esquemático Eletrônico

Código-Fonte

```
int scVerde = 10;
int scAmarelo = 11;
int scVermelho = 12;
int spVerde = 8;
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 DHT11, 2 LEDs vermelhos, 2 LEDs amarelos, 2 LEDs verdes, 1 Buzzer

É possível construir um termômetro utilizando o Kit Arduino, LEDs e um sensor de temperatura. Dependendo do valor da temperatura ambiente, ele acende n LEDs que correspondem a temperatura lida. Para ilustrar melhor, imagine um circuito com 20 LEDs onde cada LED correspondesse a 1^oC. Caso o sensor leia uma temperatura de 15^oC em uma sala, isso significa que os 15 primeiros LEDs deverão acender.
Como essa escala utiliza muitos LEDS, implemente um termômetro que utilize 6 LEDs onde cada um representa uma determinada faixa de temperatura. Para incrementar o projeto, faça com que quando o termômetro indicar uma situação crítica de temperatura no ambiente, ou seja, quando todos os LEDs estiverem acesos, um Buzzer seja acionado, indicando uma alta temperatura ambiente. Seu projeto deve seguir o seguinte padrão:

- temperatura maior que 15^oC: acenda o primeiro LED verde. Caso contrário, mantenha-o apagado.
- temperatura maior que 20^oC: acenda o segundo LED verde. Caso contrário, mantenha-o apagado.
- temperatura maior que 25^oC: acenda o primeiro LED amarelo. Caso contrário, mantenha-o apagado.
- temperatura maior que 30^oC: acenda o segundo LED amarelo. Caso contrário, mantenha-o apagado.
- temperatura maior que 40^oC: acenda o primeiro LED vermelho. Caso contrário, mantenha-o apagado.
- temperatura maior que 50^oC: acenda o segundo LED vermelho. Caso contrário, mantenha-o apagado.

Lembre-se de 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 DHT11 na porta 2. Observe a Figura 10.

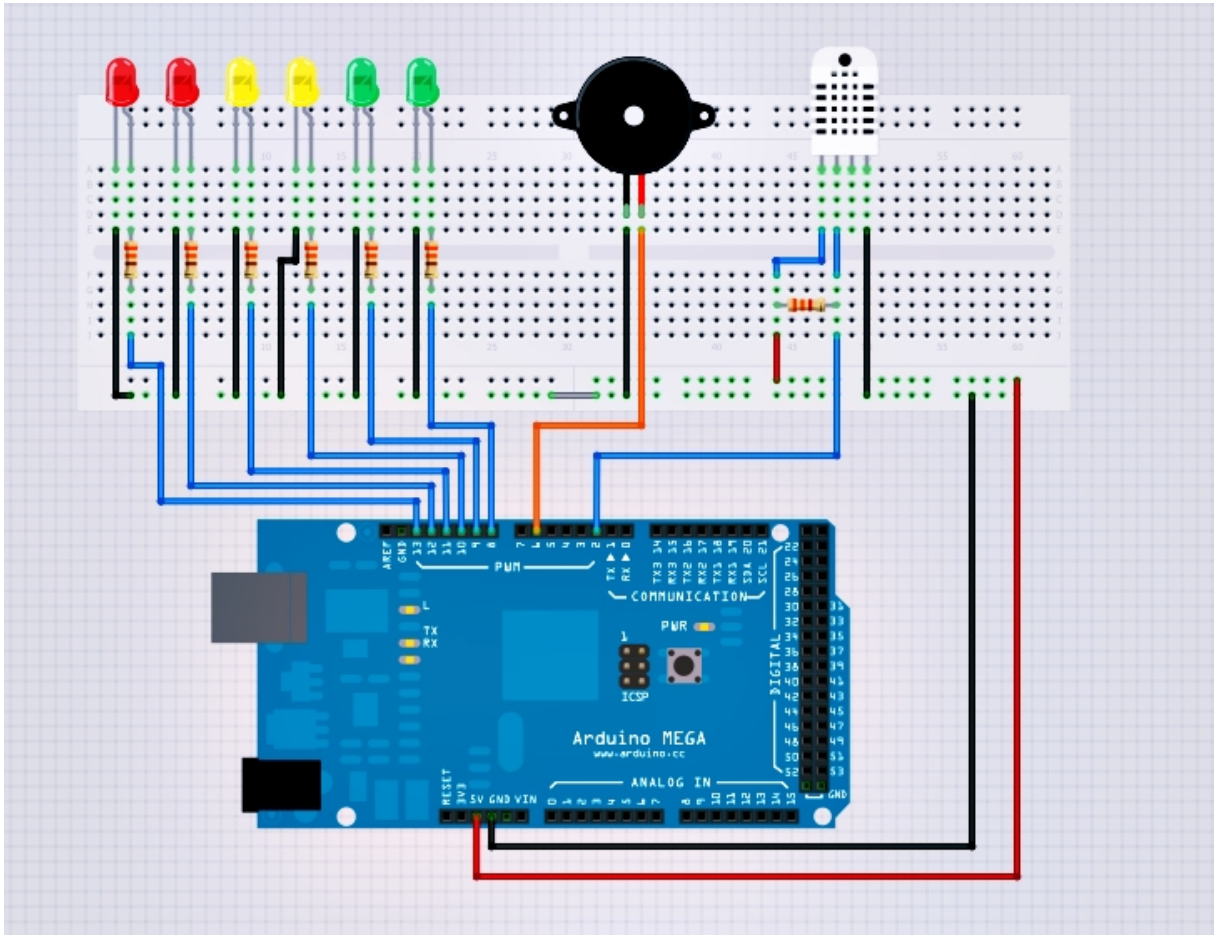

Figura 10: Montagem do Circuito

Código-Fonte

```
#include <DHT.h>

#define DHTPIN 2
#define DHTTYPE DHT11

DHT dht(DHTPIN, DHTTYPE);
float temperatura, umidade;

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

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

void loop(){

        temperatura = dht.readTemperature();
        umidade = dht.readHumidity();

        Serial.print("Umidade (%): \t");
        Serial.println(umidade);

        Serial.print("Temperatura (oC): \t");
        Serial.println(temperatura);
```

```
        if (temperatura > 15)
              digitalWrite(led1, HIGH);
        else
              digitalWrite(led1, LOW);
        if (temperatura > 20)
              digitalWrite(led2, HIGH);
        else
              digitalWrite(led2, LOW);
        if (temperatura > 25)
              digitalWrite(led3, HIGH);
        else
              digitalWrite(led3, LOW);
        if (temperatura > 30)
              digitalWrite(led4, HIGH);
        else
              digitalWrite(led4, LOW);
        if (temperatura > 40)
              digitalWrite(led5, HIGH);
        else
              digitalWrite(led5, LOW);
        if (temperatura > 50 ){
              digitalWrite(led6, HIGH);
              analogWrite(buzzer, 80)
        }
        else {
              digitalWrite(led6, LOW);
              analogWrite(buzzer, 0)
        }
        delay(1000);
}
```

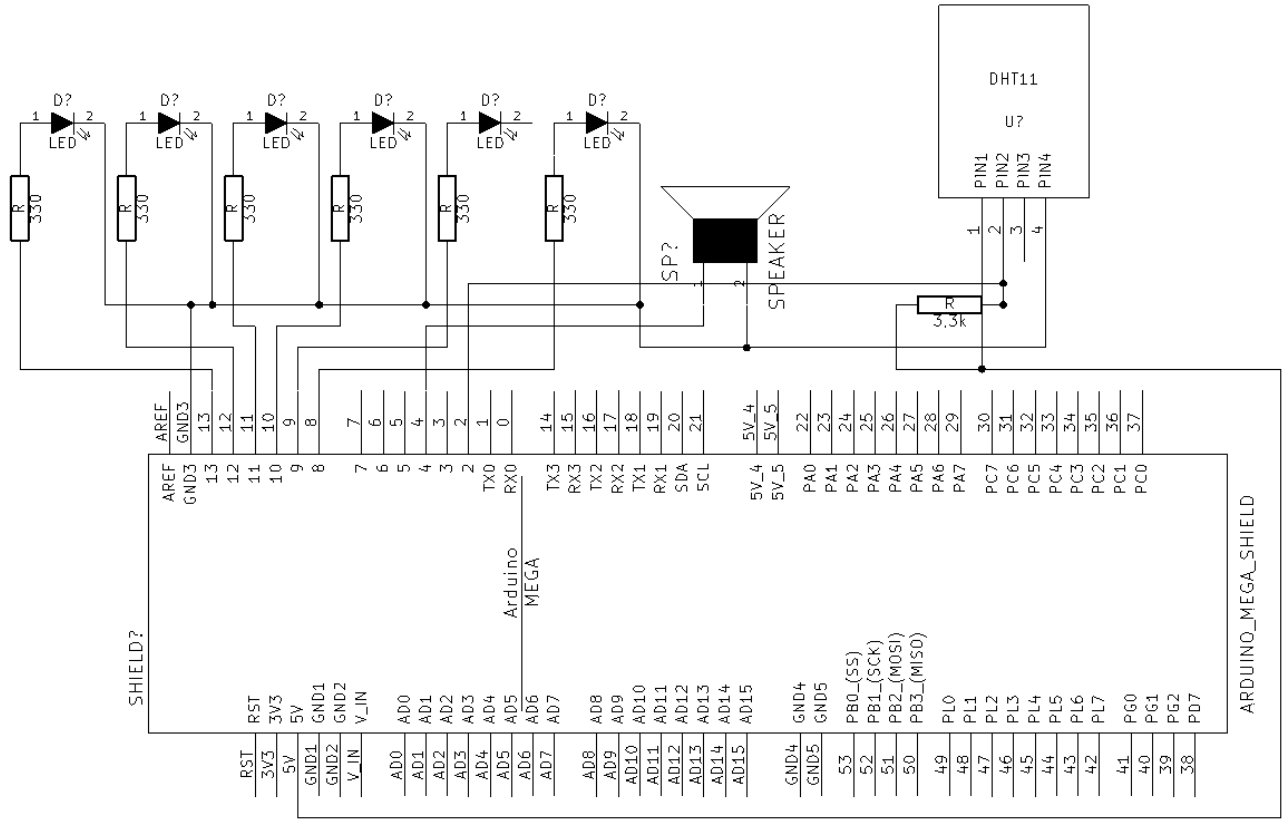

Figura 11: Esquemático Eletrônico

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 não tem capacidade 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 tem-se apenas 3 botões e sete notas musicais, cada botão vai referenciar a mais de uma nota musical, logo, assim também será com 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 as Figuras 12 e 13.

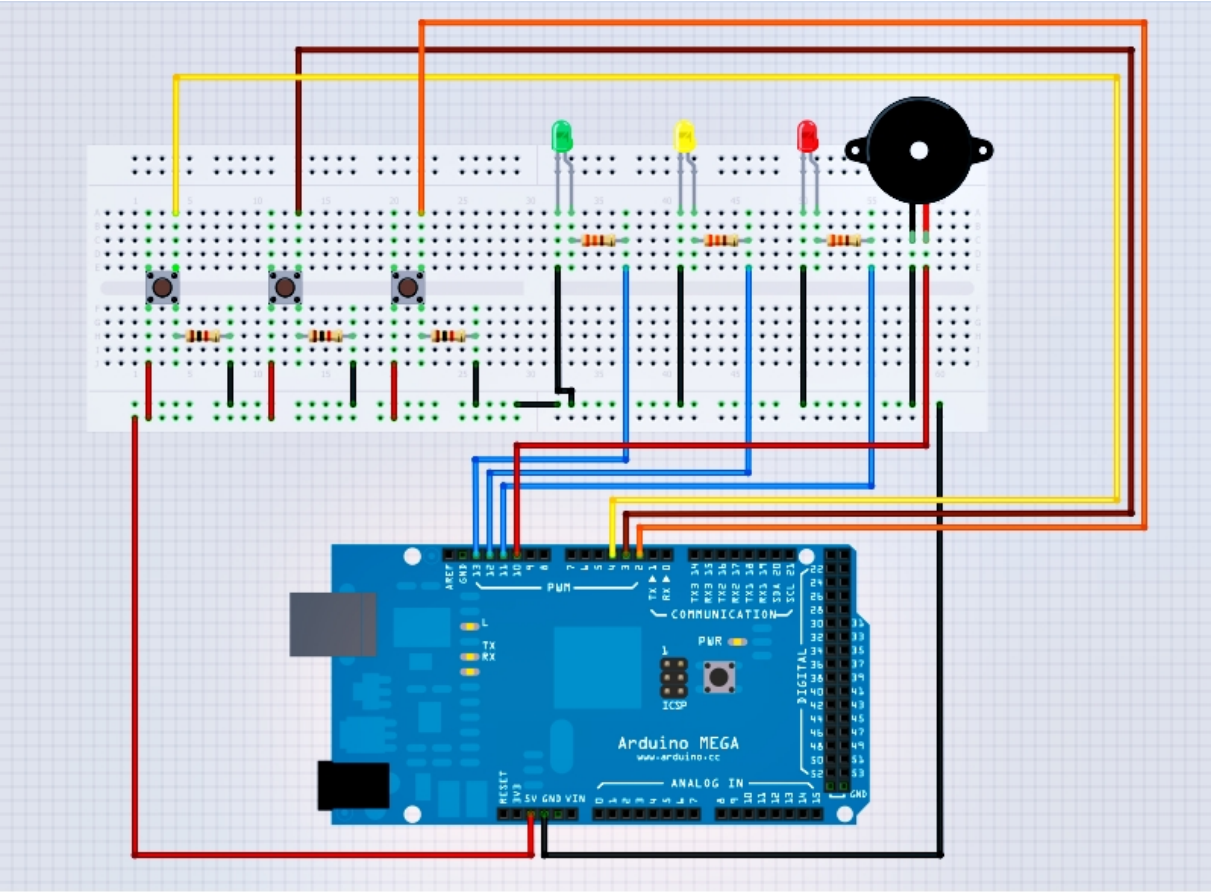

Figura 12: Montagem do Circuito

Código-fonte

```
int ledPin1 = 13;
int ledPin2 = 12;
int ledPin3 = 11;
int Botao1 = 2;
int Botao2 = 3;
int Botao3 = 4;
```

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

```
}
```

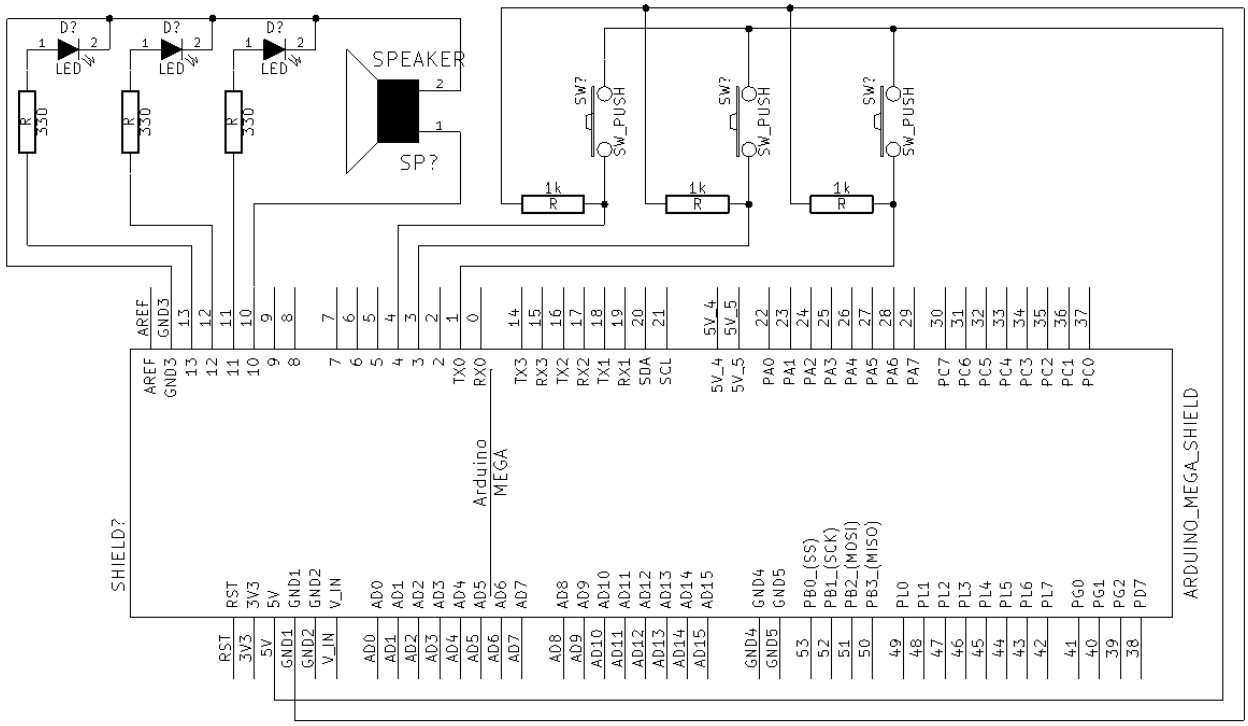

Figura 13: Esquemático Eletrônico

9 Exemplo de Aplicação 6: Alarme

Componentes: 1 Sensor Ultrassônico, *Buzzer*, 1 LED

Hoje em dia é comum encontrar sensores de distância instalados na traseira de carros para auxiliarem os motoristas na hora de estacionarem. Esses sensores detectam objetos que estão em uma determinada distância e caso o sistema 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, em residências e em escritórios para indicar a presença de alguém no ambiente. Alguns desses sensores emitem um *beep* e outros 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

Conecte o LED no pino 11 e o *Buzzer* no pino 10. Para conectar o sensor ultrassônico, observe as Figuras 14 e 15

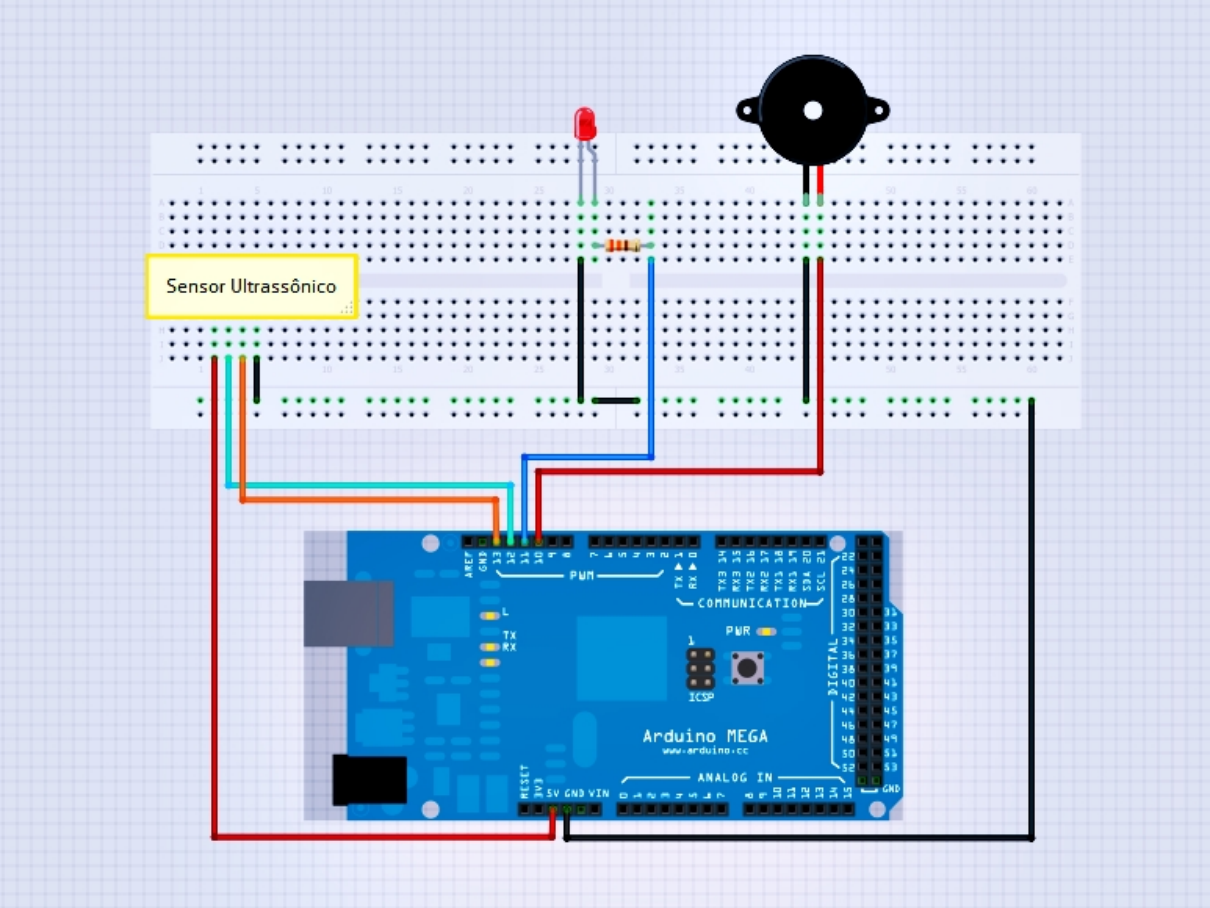

Figura 14: Montagem do Circuito

Código-fonte

```
#include "Ultrasonic.h"

int echoPin = 13;
int trigPin = 12;
int LED = 11;
int buzzer = 10;

Ultrasonic ultrasonic(trigPin, echoPin);
int distancia;

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

void loop() {
     digitalWrite(trigPin, HIGH);
     delayMicroseconds(10);
     digitalWrite(trigPin, LOW);
     distancia = ultrasonic.Ranging(CM);

     if(distancia < 20){
          digitalWrite(LED, HIGH);
          analogWrite(buzzer, 80)
     }
     else{
          digitalWrite(LED, LOW);
          analogWrite(buzzer, 0)
     }
}
```

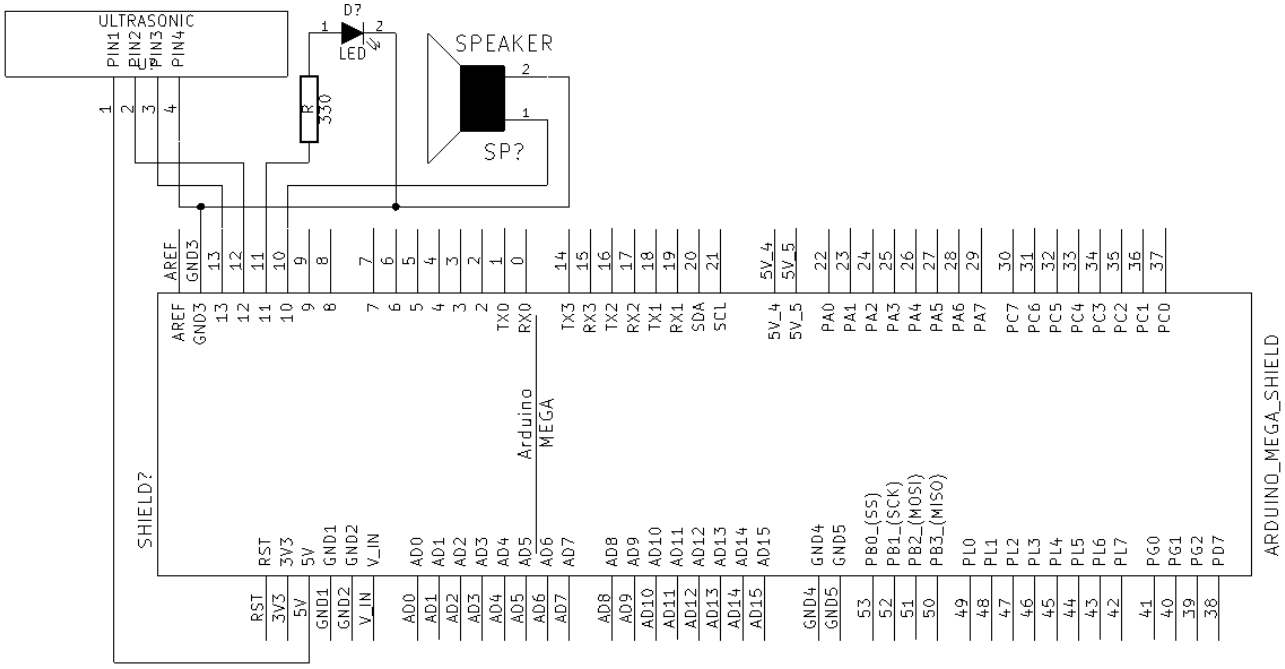

Figura 15: Esquemático Eletrônico

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 DHT11, 1 LED Alto Brilho, 1 Buzzer

Neste exemplo deve-se ter um cuidado maior, por ele ser mais complexo que os anteriores. Neste projeto, 3 LEDs (1 de cada cor) correspondem à temperatura e os outros 3 correspondem à luminosidade. Deve-se implementar o 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.

Sugestão de montagem

Conecte os LEDs nos pinos de 5 à 11, o *Buzzer* no pino 2, o sensor de temperatura DHT11 no pino 12 e o sensor de luminosidade na porta 0. Observe as Figuras 18 e 17

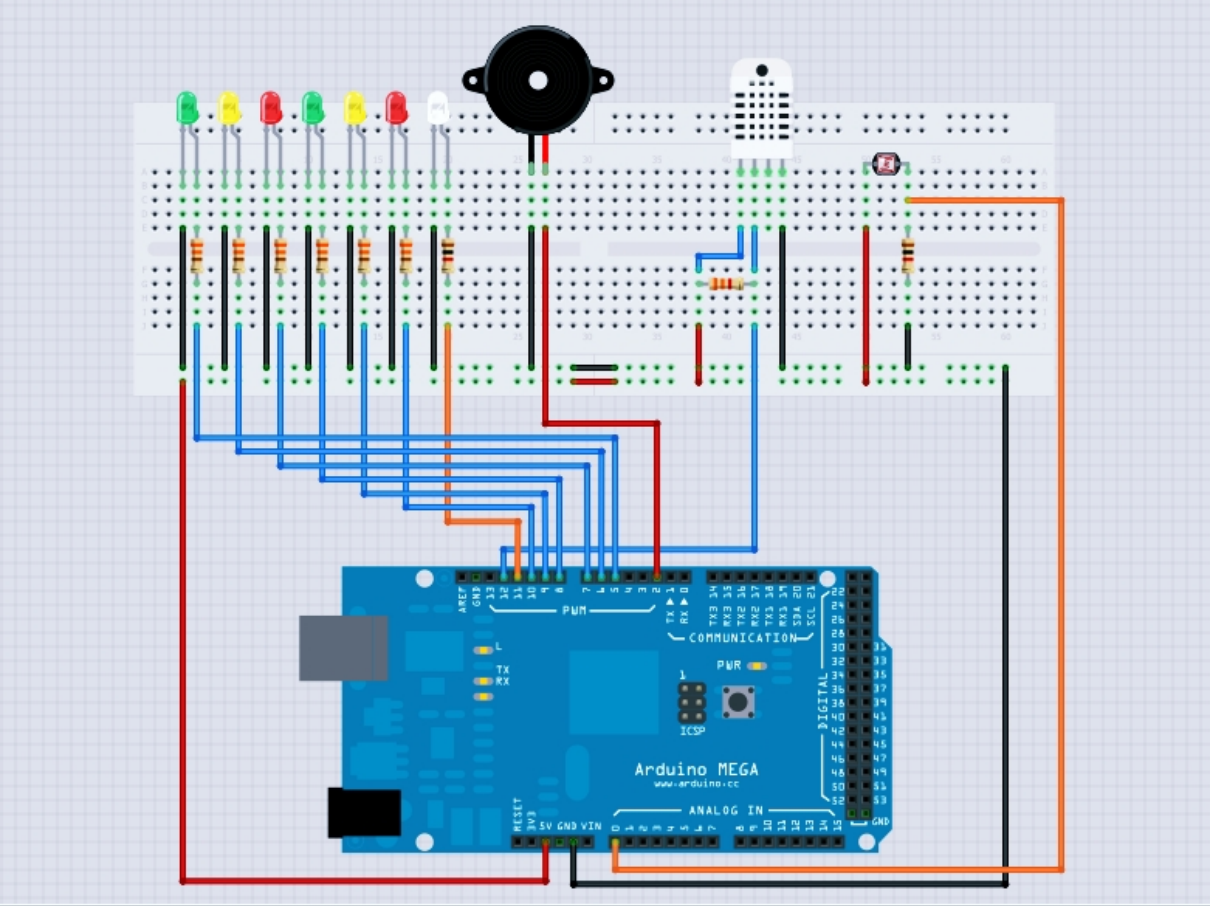

Figura 16: Montagem do Circuito

Código-fonte

```
#include <DHT.h>
#define DHTPIN 12
#define DHTTYPE DHT11
DHT dht(DHTPIN,DHTTYPE);
```

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

int ValorLDR = 0;
float ValorDHT11 = 0;

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

void loop(){
        ValorLDR = analogRead(LDR);
        ValorDHT11 = dht.readTemperature();
        Serial.print("Valor da Temperatura = ");
        Serial.println(ValorDHT11);
        if (ValorDHT11 > 10)
             digitalWrite(led1, HIGH);
        else
             digitalWrite(led1, LOW);
        if (ValorDHT11 > 20)
             digitalWrite(led2, HIGH);
        else
             digitalWrite(led2, LOW);
```

```
        if (ValorDHT11 > 30) {
             digitalWrite(led3, HIGH);
             analogWrite(buzzer, 80)
        }
        else {
             digitalWrite(led3, LOW);
             analogWrite(buzzer, 0)
        }

        Serial.print("LDR ");
        Serial.println(ValorLDR);
        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);
        }
}
```

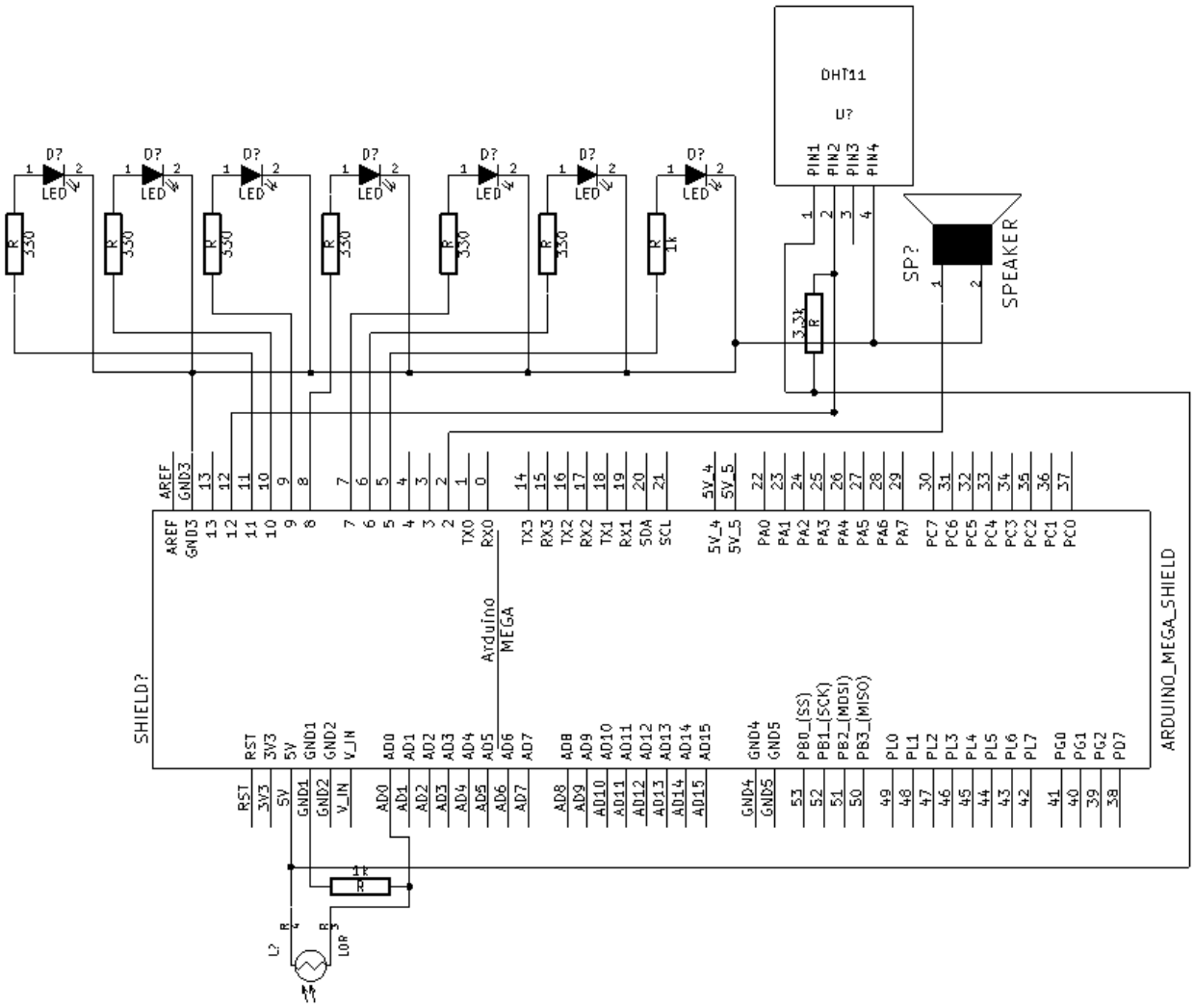

Figura 17: Esquemático Eletrônico

11 Exemplo de Aplicação 8: Portão Eletrônico

Componentes: 1 servo-motor, 1 sensor de distância ultrassônico, 1 buzzer, barra retangular de cartolina ou papelão.

Neste exemplo será utilizado um servo-motor para realizar a elevação de uma barra retangular simulando o funcionamento de um portão eletrônico. Além do servo-motor, será utilizado também um sensor de distância ultrassônico para acionar o servo-motor caso a distância seja menor que

determinado limite em centímetros. Além disso, um alarme deve soar quando o portão fechar. O funcionamento do circuito acontece da seguinte forma:

- O sensor de distância deve ser posicionado a frente (utiliza uma segunda *protoboard*) do servo-motor.
- Se a distância retornada pelo sensor for menor ou igual a determinado limite (10cm), o servo-motor deve ser acionado 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 alarme é acionado, o sensor aguarda 3 segundos e aciona o servo-motor para retornar a posição inicial (giro de 90^o para direita).

Sugestão de montagem

Siga o exemplo das Figuras 18 e 19

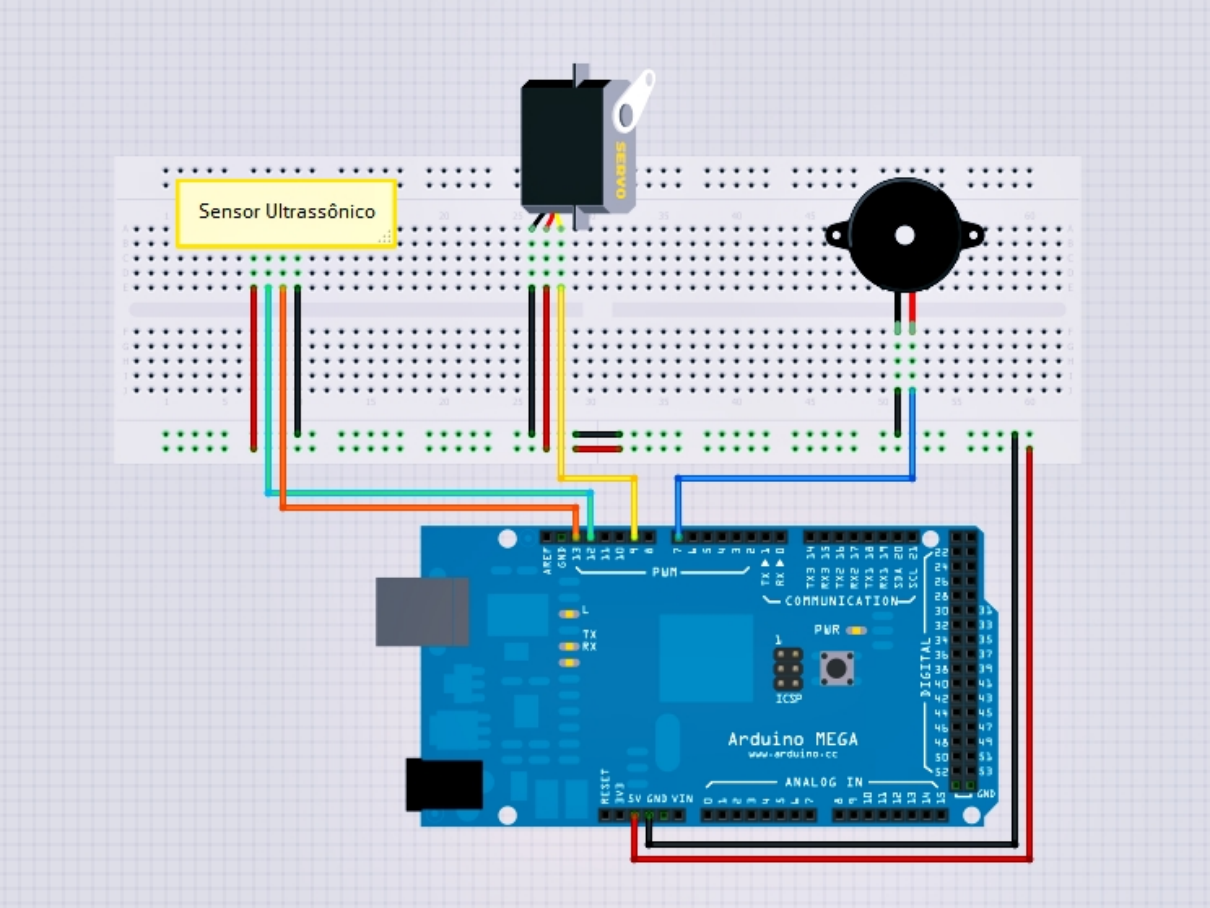

Figura 18: Montagem do Circuito

Código-fonte

```
#include <Servo.h>
#include <Ultrasonic.h>

Servo myservo;
int echoPin = 13;
int trigPin = 12;
int buzzer = 7;
int distancia;
boolean flag = false;
long microsec;
```

```
int pos;
Ultrasonic sensor(trigPin, echoPin);

void setup() {
  myservo.attach(9);
  pinMode(echoPin, INPUT);
  pinMode(trigPin, OUTPUT);
  pinMode(buzzer, OUTPUT);
  Serial.begin(9600);
  myservo.write(0);
}

void loop() {
  microsec = sensor.timing();
  distancia = sensor.convert(microsec, Ultrasonic::CM);
  Serial.print("Distância (cm) = ");
  Serial.println(distancia);

  if(distancia < 20) {
      for(pos= 0; pos < 90; pos++) {
        myservo.write(pos);
        delay(50);
      }
      flag = true;
   }
   while(distancia < 20) {
      microsec = sensor.timing();
      distancia = sensor.convert(microsec, Ultrasonic::CM);
      Serial.print("Distância (cm) = ");
      Serial.println(distancia);
      delay(30);
   }
   if(flag == true) {
      analogWrite(buzzer, 80)
      delay(1000);
      analogWrite(buzzer, 0)
      delay(3000);
      for(pos= 90; pos > 0; pos--){
         myservo.write(pos);
         delay(50);
```

```
        }
        flag = false;
    }
}
```

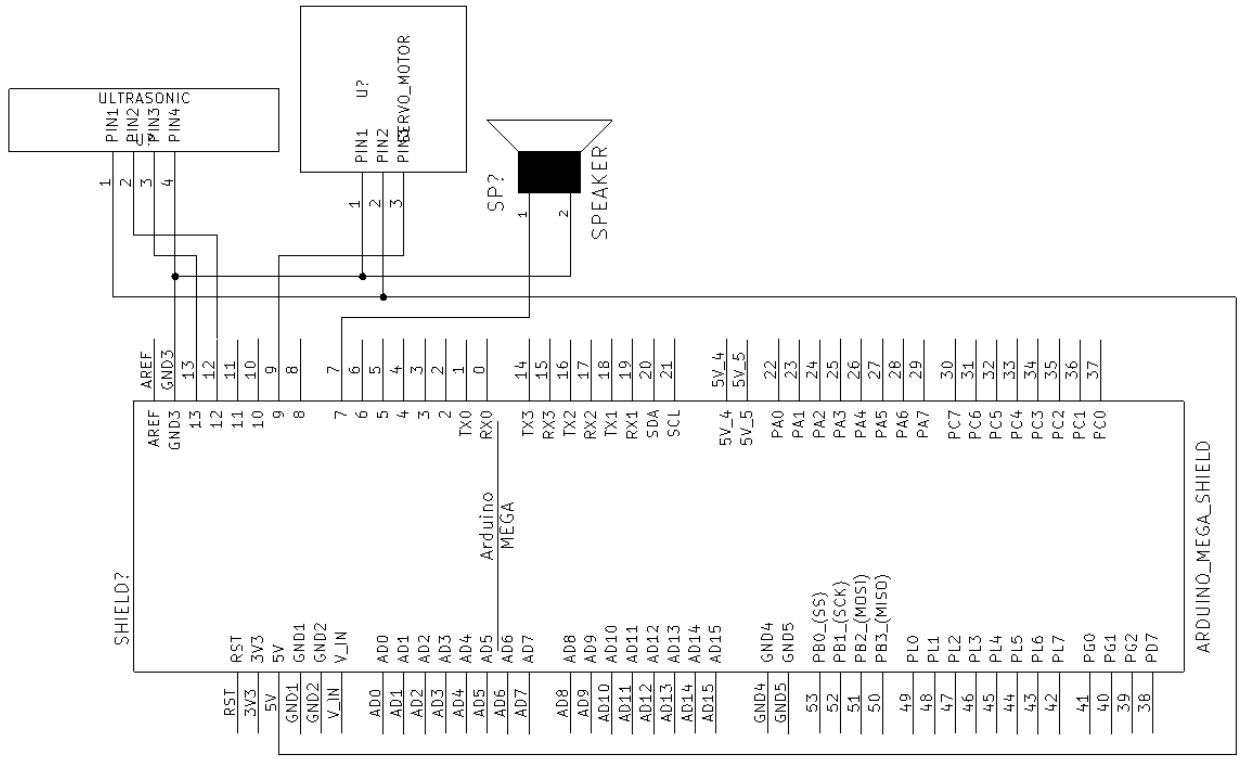

Figura 19: Esquemático Eletrônico

APÊNDICE

A Sensores e Componentes

A.1 LEDs

LED é um diodo emissor de luz (*Light Emission Diode*) que, quando alimentado corretamente, permite o fluxo de corrente elétrica em apenas um sentido. É um componente polarizado (com polo positivo e negativo) e deve ser conectado corretamente para funcionar. O terminal positivo (ânodo) é maior que o terminal negativo (cátodo). A Figura 20 mostra alguns LEDs de diferentes cores.

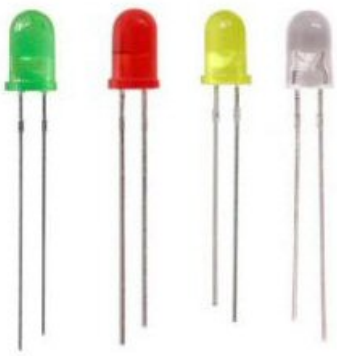

Figura 20: LEDs de várias cores

Pinagem: o terminal maior (positivo) do LED deve ser conectado em um dos terminais do resistor, que por sua vez deve ser conectado em um dos pinos da placa Arduino. O terminal menor do LED deve ser conectado ao pino GND do Arduino, como mostra a Figura 21 .

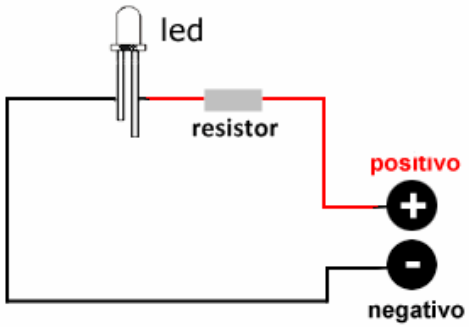

Figura 21: Representação da pinagem de um LED

Resistor adequado: Primeiramente, é preciso deixar claro que para cada cor e tamanho de LED há uma voltagem própria para que o mesmo acenda. Na maioria dos casos, LED's vermelhos, verdes e amarelos precisam de uma tensão entre 2,2 V e 2,4 V (quase constante decorrente do fato de o LED ser um diodo). Quanto ao fluxo de corrente necessária, muitos dos LED's necessitam somente de 20 mA. Tendo esses valores em mãos, é possível determinar o melhor valor de resistor para cada cor de LED. Para isso, usa-se a equação:

$$R = \frac{(V_f - V_l)}{I_l}$$

onde R é a resistência do resistor (em ohms), V_f é a tensão da fonte (em Volts), V_l é a queda de tensão para o LED em questão (em Volts) e I_l é a corrente suportada com segurança pelo LED.

A.2 Potenciômetro

Um potenciômetro é um componente com uma haste giratória, cuja função é variar a resistência. Normalmente, o valor gerado pelo mesmo é lido pelo Arduino como um valor analógico em uma entrada analógica. Uma aplicação bastante simples desse componente é controlar a frequência com que um LED pisca, por meio da variação da resistência ao girar o botão do potenciômetro.

Figura 22: Potenciômetro 10KΩ

Pinagem: Como pode-se observar na Figura 22, um potenciômetro possui 3 terminais. Um dos terminais da extremidade do potenciômetro deve ser conectado ao GND da placa e a outra extremidade à alimentação +5 V. O terminal central é conectado a um pino de entrada analógica da placa Arduino.

A.3 Push-button

Push-button (Figura 23) é um componente que conecta dois pontos no circuito ao ser pressionado (como ligar um LED ao pressioná-lo).

Figura 23: Dois *push-buttons* de 4 terminais

Pinagem:

Um dos quatros terminais do botão é conectado a um resistor que por sua vez é conectado ao GND da placa Arduino. Outro terminal é conectado ao +5 V do Arduino, e o terceiro terminal é conectado na entrada digital, que irá ler o estado atual do botão, como mostra a Firgura 24. Quando o push-button está aberto (não pressionado), a entrada do Arduino fica conectada ao GND pelo resistor e retornará o valor LOW em uma leitura. Quando o push- button, está pressionado a entrada fica conectada ao + 5V e o valor lido será HIGH.

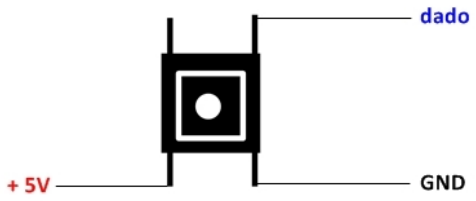

Figura 24: Pinagem do *push-button* de 4 terminais

Resistor adequado: em geral, recomenda-se utilizar resistores que estejam na faixe de 220Ω a 1 KΩ.

A.4 Buzzer 5V

Buzzer (Figura 25) é uma campainha que reproduz um beep de acordo com as variações de tensões em seus terminais.

Figura 25: *Buzzer* 5V

Pinagem:

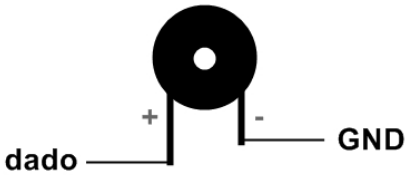

Figura 26: Pinagem de um *Buzzer* 5V

A.5 Sensor de Luminosidade LDR 5mm

Um sensor de luminosidade (*Light Dependent Resistor*) é basicamente um sensor que permite detectar a luz do ambiente variando seu valor de resistência (em Ω), dependendo da intensidade de luz em sua superfície. Eles são baratos, fáceis de adquirir em vários tamanhos e especificações, mas também são muito imprecisos. Cada sensor age de forma diferente um do outro, mesmo que sejam de um mesmo lote. Por este motivo não se pode confiar neles para se determinar níveis precisos de luz, mas são bem úteis para se determinar variações básicas de luz. No escuro sua resistência pode alcançar 1 KΩ e diminui conforme a intensidade da luz aumenta, ou seja, sua resistência diminui proporcionalmente à intensidade de luz ambiente detectada.

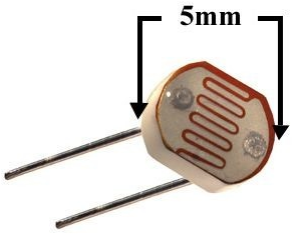

Figura 27: Sensor de Luminosidade LDR 5mm

Pinagem: conecte um dos terminais do LDR ao +5 V da placa Arduino e o outro terminal conecte ao pino analógico da placa arduino. Nesse mesmo terminal, conecte um dos terminais de um resistor de 10 KΩ, por exemplo. O outro terminal do resistor deve ser conectado ao GND da placa Arduino

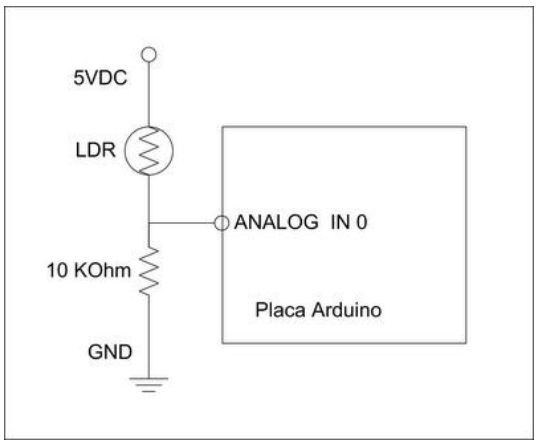

Figura 28: Esquemátcio do sensor de luminosidade

Resistor adequado: A resistência de um sensor de luminosidade varia de acordo com a marca, modelo e tamanho, não existindo um resistor padrão para ser utilizado.

A.6 Sensor de temperatura e umidade DHT11

Um sensor de temperatura e umidade faz exatamente o que o nome sugere, lê a temperatura e a umidade do ar. É um sensor básico e geralmente lento, mas é ótimo para se fazer algumas coletas básicas de dados. O sensor DHT (Figura 29) é composto de duas partes, um sensor capacitivo de umidade e um termistor. Também há um chip bastante básico dentro dele que faz algumas conversões de analógico para digital e manda um sinal digital com a temperatura e umidade. Este sinal digital é bastante fácil de ler usando qualquer microcontrolador.

Figura 29: Sensor de temperatura e umidade DHT11

Pinagem: Alguns Sensores DHT11 possuem as indicações sobre onde deve ser ligado cada terminal (como na Figura 29). Outros sensores possuem quatro terminais, dos quais apenas três são usados: os dois primeiros e o quarto terminal. O primeiro terminal é ligado na saída de +5V do Arduino. O segundo é conectado no pino de entrada de dados na placa Arduino e o quarto terminal é ligado ao GND do Arduino. A Figura 30 demonstra como conectar o sensor DHT11 com o Arduino, utilizando um resistor de 10K.

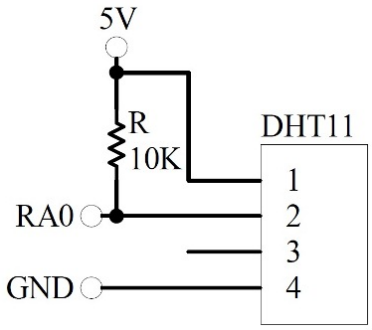

Figura 30: Sensor de temperatura e umidade DHT11 ligado a um Arduino

Resistor adequado: Um sensor de umidade e temperatura DHT utiliza, um resistor de 10 KΩ para que seu funcionamento seja adequado, podendo variar conforme o modelo.

A.7 Sensor Infravermelho - Sharp GP2Y0A21YK0F

O sensor infravermelho - Sharp GP2Y0A21YK0F (Figura 31), faz uso das leituras em infravermelho para detectar a distância do sensor até objetos, isto é, através da leitura em infravermelho é possível fazer a detecção de objetos que estejam a frente do sensor, muito útil caso se construa um robô que necessite desviar de obstáculos por exemplo.

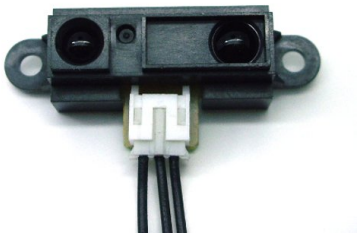

Figura 31: Sensor Infravermelho - Sharp GP2Y0A21YK0F

Pinagem: O sensor infravermelho - Sharp GP2Y0A21YK0F possui três terminais, sendo o primeiro para saída de dados, o segundo sendo o GND (ground) e o terceiro o pino de 5V, a Figura 32 mostra a ligação com uma placa Arduino.

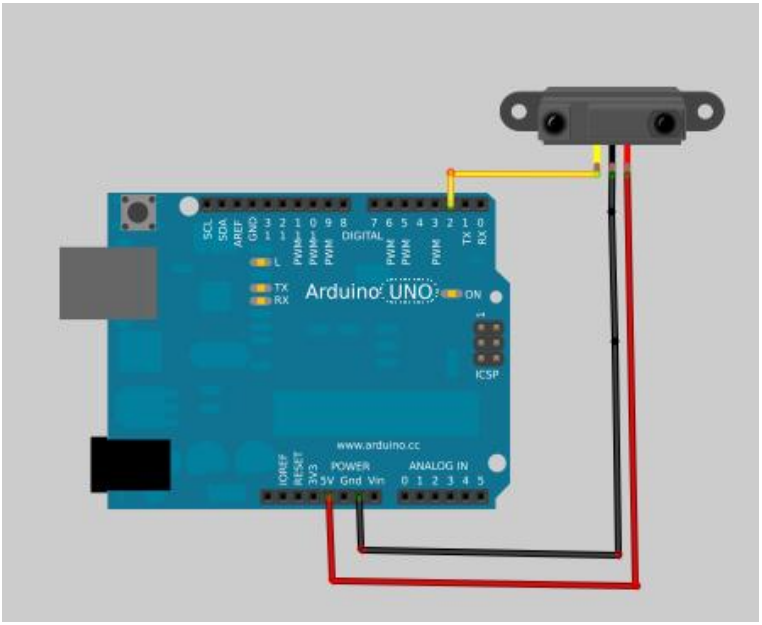

Figura 32: Sensor Infravermelho - Sharp GP2Y0A21YK0F

A.8 Servo motor

Servo motor é um motor com amplitude de angulação inferior aos demais motores, em geral, conseguindo realizar um deslocamento de 180^o. Um servo motor recebe um sinal de controle, verifica a posição que está e então se move para a posição desejada. Eles são muito mais precisos do que os demais motores para atingir a posição final desejada.

Figura 33: Servo Motor

Pinagem: Em geral, os servos-motores seguem um padrão internacional para cores: o terminal preto deve ser conectado o GND, o terminal vermelho no + 5 V e o terceiro terminal, que pode assumir diferentes cores ao variar o modelo do servo, deve ser conectado na saída PWM de dados da placa Arduino.

A.9 Display de 7 segmentos

O *Display de 7 segmentos* é um dispositivo muito útil quando se quer mostrar alguma informação numérica (Figura 34).

Pinagem: Basicamente os segmentos que estiverem com valor LOW estão apagados e os que estiverem com valor HIGH estão acesos, pois esse *display* é do tipo cátodo comum.

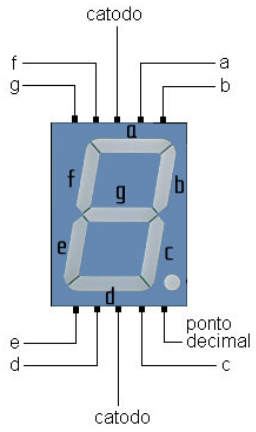

Figura 34: *Display de 7 segmentos*

Resistência: Para cada segmento do display deve-se colocar um resistor entre 220 Ω e 550 Ω para limitar a corrente elétrica através dos segmentos que são LEDs.

A.10 LCD 16x2

O Display de Cristal Líquido (*Liquid Cristal Display*) é um dispositivo utilizado para mostrar informações na forma de texto (Figura 35). Um modelo muito utilizado é o que apresenta 2 linhas e 16 colunas, onde pode-se representar até de 32 caracteres.

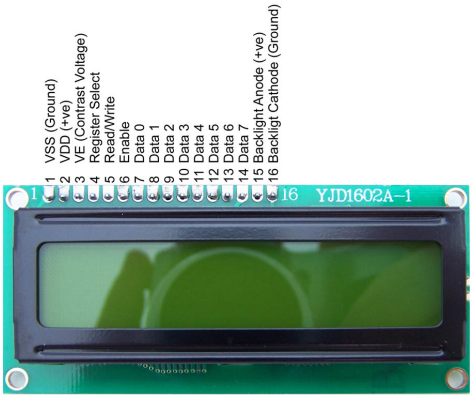

Figura 35: Descrição de cada pino de um LCD 16x2 5V

Resistência: Além da necessidade de um potenciômetro 10 KΩ, utilizado para controlar o contraste dos caracteres que são mostrados na tela do LCD, é recomendável adicionar uma resistência de 200Ω no pino 15 para limitar a corrente no LED da iluminação de fundo

Pinagem:

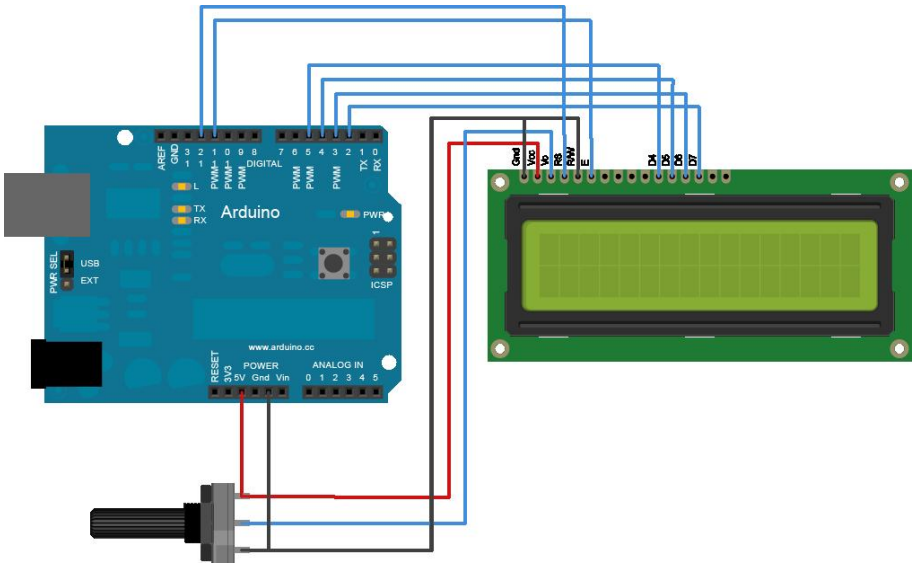

Figura 36: Pinagem de um LCD

Tabela 7: Funcionalidade de cada pino de um LCD

Pino	Função	Descrição
1	Alimentação	Terra (GND)
2	Alimentação	VCC ou +5V
3	V0	Tensão para ajuste de contraste
4	RS	Seleção: 1 – Dado. 0 – Instrução.
5	R/W	Seleção: 1 – Leitura. 0 – Escrita.
6	E	Chip select: 1 – Habilita. 0 – Desabilita
7	B0	LSB
8	B1	Barramento de Dados
9	B2	
10	B3	
11	B4	
12	B5	
13	B6	
14	B7	MSB
15	A	(quando existir) Ânodo para LED *backlight*
16	K	(quando existir) Cátodo para LED *backlight*

A.11 Resistores

A.11.1 O que são resistores?

São elementos que apresentam resistência à passagem de corrente elétrica, sendo que quanto maior a sua resistência, menor é a corrente elétrica que passa em um resistor. Os resistores geralmente possuem um formato cilíndrico e faixas coloridas que definem o seu valor em Ohms. Servem para opor-se a passagem de corrente, ficando assim certa tensão sobre o mesmo.

O valor de um resistor pode ser facilmente identificado analisando-se as cores em torno dele ou então usando um ohmímetro (instrumento de medição de resistência elétrica).

O material do resistor é uma película fina de carbono (filme), depositada sobre um pequeno tubo de cerâmica. O filme resistivo é enrolado em hélice por fora do "tubinho" até que a resistência entre os dois extremos fique tão próxima quanto possível do valor que se deseja. São acrescentados terminais (um em forma de tampa e outro em forma de fio) em cada extremo. Em seguida o resistor é recoberto com uma camada isolante e no fim suas faixas coloridas transversais são pintadas para indicar o valor da sua resistência.

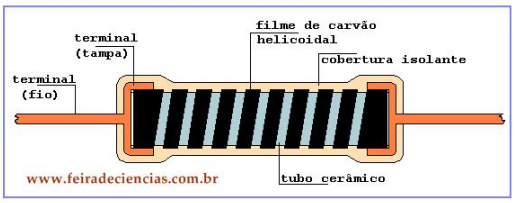

Figura 37: O resistor internamente

A.11.2 Tipo de resistores

Os resistores podem ser de resistência fixa ou variável.

- **Resistor fixo:** É um resistor que apresenta um único valor de resistência.
- **Resistor variável:** Seus valores podem ser ajustados por um movimento mecânico, ou seja, manualmente.

O potenciômetro é um tipo de resistor variável comumente utilizado para controlar o volume em amplificadores de áudio. Geralmente, é um resistor de três terminais onde a conexão central é deslizante e manipulável. Se todos os três terminais são usados, ele atua como um divisor de tensão

A.11.3 Resistores em série e em paralelo

Resistores em série

Em um circuito em série constatam-se as seguintes propriedades:

- A corrente que passa por todos os componentes é a mesma;
- A soma das tensões sobre todos os componentes deve ser igual à tensão total aplicada;
- A resistência total da associação em serie é igual à soma das resistências dos componentes individuais.

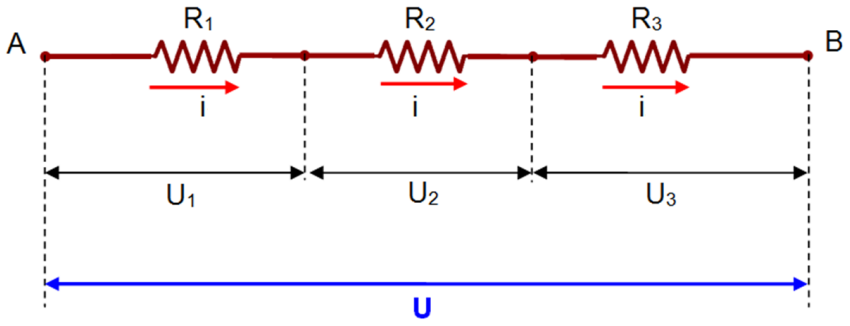

Figura 38: Associação de resistores em série

Resistores em paralelo

Em um circuito paralelo constatam-se as seguintes propriedades:

- Todos os componentes recebem a mesma tensão elétrica;
- A soma de todas correntes nos componentes individuais deve ser igual à corrente total;
- A resistência total da associação é calculada pelo quociente entre o produto das resistências individuais e a soma delas (CUIDADO: isso vale só para 2 resistores em paralelo

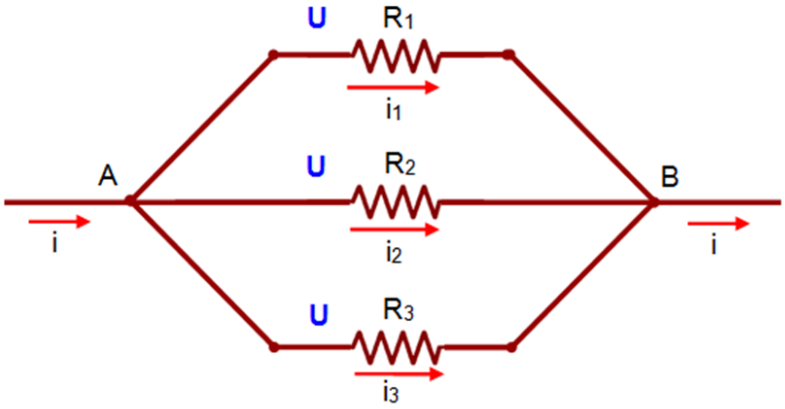

Figura 39: Associação de resistores em paralelo

A.11.4 Código de Cores

Por ter geralmente um tamanho muito reduzido, é inviável imprimir nos resistores as suas respectivas resistências. Optou-se então pelo código de cores, que consiste em faixas coloridas no corpo do resistor.

As primeiras três faixas servem para indicar o valor nominal de sua resistência e a última faixa, a porcentagem na qual a resistência pode variar seu valor nominal (tolerância).

Para encontrar a resistência de um resistor e sua tolerância, usa-se os dados da tabela da Figura 40 e a seguinte equação:

$$Resistência = (10 \ x \ faixa \ 1 + faixa \ 2) \ x \ 10^{(faixa3)} \pm \% \ de \ tolerância$$

Obs. 1: Na faixa 3, são permitidos valores somente até 7, o dourado passa a valer -1 e o prateado -2.

Obs. 2: A ausência de uma quarta faixa, indica uma tolerância de 20 %.

Alguns *websites* disponibilizam uma página que pode-se inserir as cores de um resistor (em sua devida ordem) e ele mostra qual a sua resistência, como na Figura 41

Cores	Valores			Multiplicadores	Tolerência
	Faixa 1	Faixa 2	Faixa 3		
Prata	-	-	-	0,01	10%
Ouro	-	-	-	0,1	5%
Preto	0	0	0	1	-
Marrom	1	1	1	10	1%
Vermelho	2	2	2	100	2%
Laranja	3	3	3	1000	-
Amarelo	4	4	4	10000	-
Verde	5	5	5	100000	-
Azul	6	6	6	1000000	-
Violeta	7	7	7	-	-
Cinza	8	8	8	-	-
Branco	9	9	9	-	-
Nenhuma	-	-	-	-	20%

Figura 40: Tabela de Cores e Valores das faixas de resistores

Figura 41: Página da web <http://www.areaseg.com/sinais/resistores.html>

B Descrição do funcionamento de uma *protoboard*

A *protoboard* (ou *breadboard*, ou ainda, matriz de contatos) é uma base para construção de protótipos eletrônicos. Ela é muito utilizada, pois não requer solda. Isto torna-a mais fácil de usar para criação de protótipos. A utilização de uma protoboard torna possível a construção de circuitos mais complexos utilizando o Arduino

A ligação de circuitos é feita através de jumpers [ver apêndice C.15] (pequenos fios), que são utilizados para ligar temporariamente componentes eletrônicos na protoboard. Normalmente, uma *protoboard* é formada por quatro matrizes, mas este número pode variar Figura 42

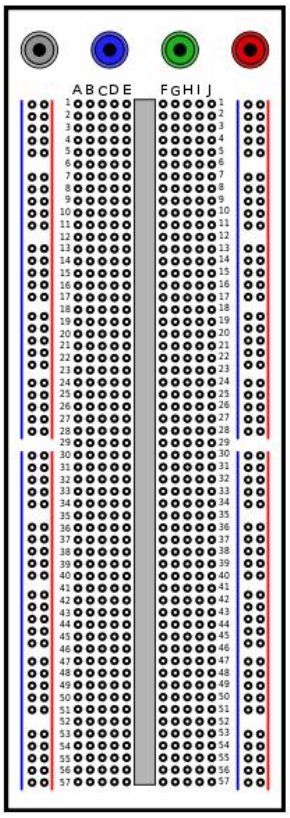

Figura 42: Exemplo de uma *protoboard*

A *protoboard* é composta de dois tipos de matrizes principais: uma com duas colunas, chamada na Figura 50 de matriz 2, e outra com cinco colunas, chamada de matriz 5. Elas diferem no modo de interconexão dos contatos elétricos.

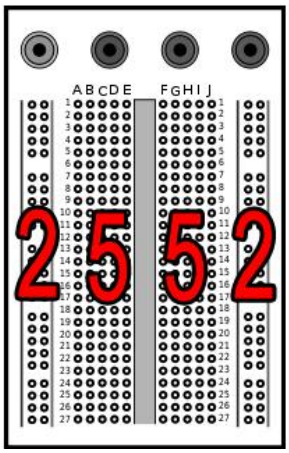

Figura 43: *protoboard* - divisão por matrizes

A matriz 2 geralmente é reservada para ligação dos pinos de alimentação +5 V e GND, e a matriz 5 utilizada para conectar componentes do circuito. A matriz 2 tem sua transmissão de coluna em coluna, enquanto a matriz 5 tem a transmissão de corrente de linha em linha.

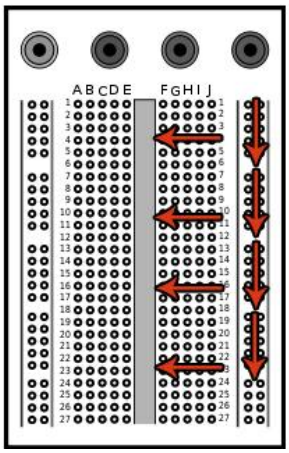

Figura 44: *protoboard* - transmissão nas matrizes

C Glossário

C.1 ASCII *(American Standard Code for Information Interchange)*

O Código padrão Americano para o Intercâmbio de Informação (Figura 45 é uma codificação numérica de caracteres capaz de representar 256 caracteres, dos quais 128 são padronizados. Até a década de 60, a maioria dos sistemas computacionais tinha seu próprio código alfanumérico, usado para representar números, letras, símbolos ou até comandos. Por causa disso, foi proposto um código comum, facilitando a comunicação entre os computadores, e assim, permitindo a troca de informações entre máquinas de diferentes fabricantes. O código ASCII original é baseado no alfabeto inglês e utiliza 7 bits representando 128 símbolos. A versão estendida utiliza 8 bits e representa mais 128 caracteres não ingleses. Nota-se que 33 caracteres são chamados caracteres de controle e são capazes de controlar funções ou equipamentos.

C.2 Biblioteca SPI

A Biblioteca SPI permite a comunicação com dispositivos SPI, tendo o Arduino como o dispositivo mestre. Pelo fato da conexão SPI sempre haver um dispositivo mestre (tendo como exemplo o Arduino), existem linhas comuns a todos os dispositivo:

- **Master In Slave Out (MISO)** é a linha que envia dados para o mestre;
- **Master Out Slave In (MOSI)** é a linha que o mestre usa para enviar dados aos "escravos";
- **Serial Clock (SCK)** é através dessa linha que o mestre fornece pulsos de clock para a sincronizar a transmissão de dados;
- **Slave Select (SS)** é pino existente em cada periférico que pode ser usado pelo mestre para habilitá-lo ou desabilita-lo, e evitar transmissões falsas devido ao ruído na linha. Em outras palavras, quando o SS está em LOW, o escravo é habilitado, e quando está em HIGH, o escravo é desabilitado. Isto permite que vários dispositivos compartilhem o mesmo MISO, MOSI e SCK.

C.3 *Bootloader*

Boot (inicializar) e *loader* (carregador) é o termo em inglês para o processo de iniciação que carrega um sistema quando uma máquina é ligada

Binário	Decimal	Hexa	Glifo
0010 0000	32	20	
0010 0001	33	21	!
0010 0010	34	22	"
0010 0011	35	23	#
0010 0100	36	24	$
0010 0101	37	25	%
0010 0110	38	26	&
0010 0111	39	27	'
0010 1000	40	28	(
0010 1001	41	29	)
0010 1010	42	2A	*
0010 1011	43	2B	+
0010 1100	44	2C	,
0010 1101	45	2D	-
0010 1110	46	2E	.
0010 1111	47	2F	/
0011 0000	48	30	0
0011 0001	49	31	1
0011 0010	50	32	2
0011 0011	51	33	3
0011 0100	52	34	4
0011 0101	53	35	5
0011 0110	54	36	6
0011 0111	55	37	7
0011 1000	56	38	8
0011 1001	57	39	9
0011 1010	58	3A	:
0011 1011	59	3B	;
0011 1100	60	3C	<
0011 1101	61	3D	=
0011 1110	62	3E	>
0011 1111	63	3F	?

Binário	Decimal	Hexa	Glifo
0100 0000	64	40	@
0100 0001	65	41	A
0100 0010	66	42	B
0100 0011	67	43	C
0100 0100	68	44	D
0100 0101	69	45	E
0100 0110	70	46	F
0100 0111	71	47	G
0100 1000	72	48	H
0100 1001	73	49	I
0100 1010	74	4A	J
0100 1011	75	4B	K
0100 1100	76	4C	L
0100 1101	77	4D	M
0100 1110	78	4E	N
0100 1111	79	4F	O
0101 0000	80	50	P
0101 0001	81	51	Q
0101 0010	82	52	R
0101 0011	83	53	S
0101 0100	84	54	T
0101 0101	85	55	U
0101 0110	86	56	V
0101 0111	87	57	W
0101 1000	88	58	X
0101 1001	89	59	Y
0101 1010	90	5A	Z
0101 1011	91	5B	[
0101 1100	92	5C	\
0101 1101	93	5D	]
0101 1110	94	5E	^
0101 1111	95	5F	_

Binário	Decimal	Hexa	Glifo
0110 0000	96	60	`
0110 0001	97	61	a
0110 0010	98	62	b
0110 0011	99	63	c
0110 0100	100	64	d
0110 0101	101	65	e
0110 0110	102	66	f
0110 0111	103	67	g
0110 1000	104	68	h
0110 1001	105	69	i
0110 1010	106	6A	j
0110 1011	107	6B	k
0110 1100	108	6C	l
0110 1101	109	6D	m
0110 1110	110	6E	n
0110 1111	111	6F	o
0111 0000	112	70	p
0111 0001	113	71	q
0111 0010	114	72	r
0111 0011	115	73	s
0111 0100	116	74	t
0111 0101	117	75	u
0111 0110	118	76	v
0111 0111	119	77	w
0111 1000	120	78	x
0111 1001	121	79	y
0111 1010	122	7A	z
0111 1011	123	7B	{
0111 1100	124	7C	|
0111 1101	125	7D	}
0111 1110	126	7E	~

Figura 45: Tabela ASCII

C.4 *Buffer*

Região de memória temporária usada escrita e leitura de dados. Sua finalidade é a compatibilização da comunicação entre dispositivos com velocidades diferentes ou variáveis. Os *Buffers* podem ser implementados por software ou hardware e no contexto da ciência da computação pode ser traduzido como retentor

C.5 *Case Sensitive*

É um termo da língua inglesa que indica que há distinção entre letras maiúsculas e minúsculas. Por exemplo, quando usamos palavras *case sensitive*, "Arduino" é diferente de "arduino", pois apesar de terem as mesmas letras, não são iguais no que diz respeito a maiúsculas e minúsculas.

C.6 Circuito Impresso

Os circuitos impressos foram criados em substituição às antigas pontes de terminais onde se fixavam os componentes eletrônicos, em montagem conhecida no jargão de eletrônica como montagem "aranha", devido a aparência final que o circuito tomava, principalmente onde existiam válvulas eletrônicas e seus múltiplos pinos terminais do soquete de fixação.

O circuito impresso consiste de uma placa de fenolite, fibra de vidro, fibra de poliéster, filme de poliéster, filmes específicos à base de diversos polímeros, etc, que possuem a superfície coberta numa ou nas duas faces por uma fina película de cobre, prata, ou ligas à base de ouro, níquel entre outras, nas quais são desenhadas pistas condutoras que representam o circuito onde serão fixados os componentes eletrônicos.

Figura 46: uma placa de circuito impresso para arduino

C.7 *Clock*

É um sinal gerado por um oscilador à cristal que sincroniza os dispositivos para realizarem suas operações, fazendo com que todos os componentes do circuito realizam suas tarefas simultaneamente, ou em ordem temporal. Esse oscilador a cristal é um circuito que utiliza a ressonância de um cristal em vibração para criar o sinal de clock com uma frequência exata e estável. O sinal do *clock* é uma onda quadrada onde os níveis lógicos 1 e 0 ocorrem alternadamente no tempo. A transição do sinal de zero para um (subida) é chamada "*rising edge*", e a transição de um para zero (descida) "*falling edge*". Alguns dispositivos eletrônicos são acionados na subida e outros na descida. A frequência (100kHz, 16MHz, e outras) indica a velocidade com que o sinal varia no tempo. Por exemplo, 16MHz são 16000000 *rising edge* e *falling edge* em um segundo.

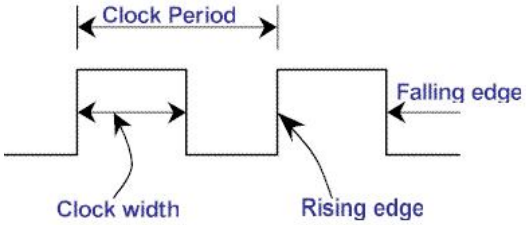

Figura 47: representação do clock

C.8 Complemento de 2

Antes de explicar o complemento de 2, deve-se definir o complemento de 1, que é obtido trocando os bits zeros por uns e vice-versa de um número binário.

O complemento de 2 de um número binário é formado tomando-se o complemento de 1 do número e adicionando-se 1 na posição do bit menos significativo (mais a direita). Por exemplo, a representação em complemento de 2 de 101011 é 010101, pois trocando os bits de 101011, tem-se 010100. Adicionado 1 tem-se 010101

C.9 Entrada/Saída digital

São interfaces de hardware que conseguem interpretar e/ou gerar sinais digitais (não analógicos).

Sinais digitais são sinais que podem assumir um número finito de valores (dois no caso dos sistemas binários). Sinais analógicos são sinais que podem assumir infinitos valores entre dois

limites.

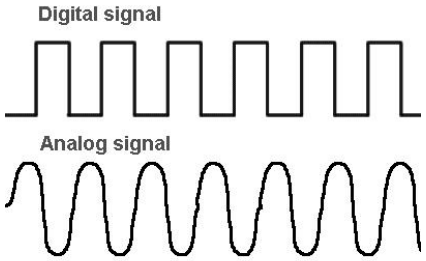

Figura 48: representação de sinal digital e sinal analógico

C.10 FTDI

É um chip que converte sinais RS-232 TTL para sinais USB. São usados em circuitos com microcontroladores, como o Arduino, permitindo a comunicação com computadores. FTDI é a sigla de *Future Technology Devices International*, que é uma empresa da Escócia que fabrica esses dispositivos semicondutores.

C.11 Fusível

Fusíveis são dispositivos protetores que são utilizados para evitar que, em caso de curto-circuito ou sobrecargas, um circuito venha sofrer danos. Basicamente um fusível funciona como o "elo mais fraco de uma corrente", fundindo e abrindo quando a corrente elétrica no circuito ultrapassa certo limite. Os fusíveis comuns usados em eletricidade e eletrônica constam de um pedaço de fio mais fino ou de menor ponto de fusão que o restante do circuito, sendo ligados em série com o mesmo.

Figura 49: representação de um Fuísvel

Quando a corrente no circuito ultrapassa certo valor, determinado pelas características do fusível, o fio rompe-se interrompendo sua circulação e evitando assim que danos possam ocorrer.

C.12 Impedância

Impedância elétrica é a medida da capacidade de um circuito de resistir ao fluxo de uma determinada corrente elétrica alternada quando se aplica uma tensão alternada através dos seus terminais.

C.13 ICSP

In-Circuit Serial Programming é um método de gravação de dispositivos programáveis, muito eficiente. ICSP significa que o dispositivo programável pode ser programado "no circuito", ou seja, é montado o circuito na placa e em seguida programa o dispositivo através de interface serial.

C.14 Jack

Jack é a parte de um conector onde se insere *plug*. O jack é o conector fêmea e o *plug* o conector macho.

C.15 *Jumpers*

É um fio condutor que tem a função de interligar dois pontos no circuito. A corrente elétrica passa pelo fio, que nada mais é do que uma continuação do circuito.

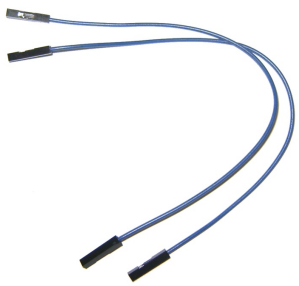

Figura 50: Dois *Jumpers*

C.16 Memória

São dispositivos eletrônicos que armazenam dados. As memórias se dividem se classificam se são voláteis ou não.

C.16.1 Memórias voláteis

Memórias Voláteis são as que necessitam de energia para manter a informação armazenada. São fabricadas com base em duas tecnologias: dinâmica e estática.

- A **memória dinâmica (DRAM)**, também conhecida como memória RAM (*Randomic Acess Memory* - memória de acesso aleatório), é a mais barata e, portanto, a mais utilizada em computadores. Sua tecnologia se baseia em acessos aos registros (local dos dados armazenados) de qualquer endereço, diferente das de acesso sequencial, que exigem que qualquer acesso seja feito a iniciar pelo primeiro endereço e, sequencialmente, vai "pulando" de um em um até atingir o objetivo. O nome dinâmica é referente à tecnologia utilizada para armazenar dados e não à forma de acessá-los. Toda vez que for acessar a memória, para escrita ou para leitura, cada célula dessa memória é atualizada. Se ela tem 1 ou 0 armazenado, sua valor será recarregado. Este procedimento é chamado de refresco de memória, em inglês, *refresh*.
- A **memória estática (SRAM)** não necessita ser analisada ou recarregada a cada momento. Fabricada com circuitos eletrônicos conhecidos como *latch*, guardam a informação por todo o tempo em que estiver a receber alimentação.

C.16.2 Memórias não voláteis

Memórias não voláteis são aquelas que guardam todas as informações mesmo sem energia. Como exemplos, citam-se as memórias conhecidas por ROM, FeRAM e FLASH, bem como os dispositivos de armazenamento em massa, disco rígido, CDs e disquetes.

As memórias somente para leitura, do tipo **ROM**, permitem o acesso aleatório e são conhecidas pelo fato de o usuário não poder alterar o seu conteúdo. Para gravar uma memória deste tipo são necessários equipamentos específicos. Dentre as memórias do tipo ROM destacam-se a **ROM (*Read Only Memory* - memória somente de leitura)**, gravada na fábrica uma única vez, **PROM (*Programable Read Only Memory* - memória programável somente de leitura)**, gravada pelo usuário uma única vez, **EPROM (*Erasable Programable Read Only Memory* - memória programável e apagável somente de leitura)**, pode ser gravada ou regravada por meio de um equipamento que fornece as voltagens adequadas em cada pino (Para

apagar os dados nela contidos, basta iluminar o chip com raios ultravioleta. Isto pode ser feito através de uma pequena janela de cristal presente no circuito integrado) e **EEPROM (*Electrically Erasable Programable Read Only Memory* - memória programável e apagável eletronicamente somente de leitura)** que pode ser gravada, apagada ou regravada utilizando um equipamento que fornece as voltagens adequadas em cada pino.

A **Memória Flash** é anterior a **FeRAM**, mas é uma variação do tipo Eprom. A principal diferença entre elas é que na Eprom as escritas são feitas byte a byte, enquanto na flash são feitas em blocos. Estas se tornaram muito populares por dois motivos: a utilização de dispositivos de armazenamento removíveis como os chamados pen drives, a aplicação em equipamentos de som que reproduzem música no formato MP3 e os cartões de memória das câmeras digitais. Os dados armazenados neste tipo de memória permanecem ali sem a necessidade de alimentação. Sua gravação é feita em geral através da porta USB que fornece 5 Volts para alimentação.

As memórias de massa podem armazenar grande quantidade de informação e têm tido seu tamanho reduzido a cada dia. O disco rígido é o meio mais comum neste tipo de memória, mas os disquetes ainda ocupam uma pequena parcela do mercado. Não é tão rápida como a memória flash mas já é possível utilizá-la em equipamentos de reprodução de música e filmes como os portáteis que reproduzem videoclipes de música em vários formatos, como MPEG.

C.17 MSB/LSB

O **MSB** *(Most Significant Bit)* e **LSB** *(Least Significant Bit)* são uma forma de se referir aos bits mais e menos significativo de um número binário, aqueles cuja potência decimal são o maior e o menor valor. O MSB é o bit mais a esquerda e o LSB é o bit mais a direita.

Como por exemplo, na Figura 51 tem-se o número binário sem sinal 10101110 que tem como MSB o 1 (quando transformado em decimal é multiplicado por 2 elevado a sétima potência) e como LSB o 0, que é multiplicado por 2 elevado a zero.

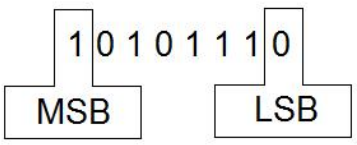

Figura 51: Exemplo de MSB/LSB

C.18 *Open-Source*

A definição do *Open-Source* foi criada pela *Open-Source Iniciative (OSI)*a partir do texto original da *Debian Free Software Guidelines (DFSG)* e determina que um programa de código aberto deve garantir: distribuição livre, o código fonte, criação de trabalhos derivados livremente, a integridade do autor do código fonte, a não discriminação contra pessoas ou grupos, a não discriminação contra áreas de atuação, a distribuição da licença, que a licença não restrinja outros programas e neutra em relação a tecnologia.

Open Source Hardware

Esse termo usado pelo arduino divide muitos dos princípios e abordagens dos *softwares open source*. Em particular, os criadores do arduino acreditam que as pessoas podem estudar seu hardware para saber como funciona, fazer mudanças e divulgar essas mudanças. Para facilitar isso, eles liberam todos os arquivos de projeto originais (*Eagle CAD*) do hardware do arduino. Estes arquivos estão sobre a licença que permite a distribuição pessoal e comercial de trabalhos derivados, desde que creditando o Arduino e liberando seus projetos sobre a mesma licença. O programa do Arduino também é *Open Source*.

C.19 *Processing*

Processing é uma linguagem de programação de código aberto e ambiente de desenvolvimento integrado (IDE), construído para as artes eletrônicas e comunidades de design visual com o objetivo de ensinar noções básicas de programação de computador em um contexto visual e para servir como base para cadernos eletrônicos. O projeto foi iniciado em 2001 por Casey Reas e Ben Fry, ambos ex-membros do Grupo de Computação do MIT Media Lab. Um dos objetivos declarados do *Processing* é atuar como uma ferramenta para não-programadores iniciarem com a programação, através da satisfação imediata de um feedback visual. A linguagem tem por base as capacidades gráficas da linguagem de programação Java, simplificando algumas características e criando outras

C.20 PWM

A Modulação por largura de pulso (MLP) - mais conhecida pela sigla em inglês "PWM" *(Pulse-Width Modulation)* - é uma forma de transmitir dados analógicos por um meio digital. O controle digital é usado para criar uma onda quadrada, que fica alternando entre nível lógico um e nível lógico zero. Este padrão "um-zero"pode representar tensões entre permanente em um

(5V) e em zero (0V), trocando a porção de tempo que o sinal fica em um contra o tempo que o sinal fica em zero. A duração em que o sinal fica em um é chamada de largura de pulso. Para conseguir valores analógicos variados, deve- se variar essa largura de pulso.

C.21 *Shields*

Shields são placas que podem ser conectados em cima do Arduino estendendo as suas capacidades. Os *shields* seguem a mesma filosofia que o Arduino: são fáceis de montar e baratos para construir

C.22 SPI *(Serial Peripheral Interface)*

Serial Peripheral Interface ou SPI é um protocolo que permite a comunicação do microcontrolador com diversos outros componentes, formando uma rede. Em modo "escravo", o microcontrolador comporta-se como um componente da rede, recebendo pulsos de clock. Em modo "mestre", o microcontrolador gera pulsos de clock e deve ter um pino de entrada/saída para habilitação de cada periférico.

C.23 TWI *(Two-Wire Interface)*

Conhecida também por **I2C** *(Inter-Integrated Circuit)*, é chamado de TWI por questão de direitos autorais. O TWI é uma forma de comunicação serial entre dispositivos de baixa velocidade. Essa comunicação utiliza duas linhas abertas:

- **SDA** *Serial Data Line*
- **SCL** *Serial Clock*

C.24 UART

É um CI (circuito integrado) que converte dados paralelos para a forma serial e vice-versa. UART é um acrônimo (sigla) de *Universal Asynchrounous Receiver/Transmiter* que significa Receptor/Transmissor Universal Assíncrono. Sinais em paralelo são vários sinais chegando ao mesmo tempo nas portas de entradas dos dispositivos (pinos), porém não é possível transmiti-los todos juntos em um barramento, por isso são convertidos na forma serial que nada mais é dos que esses sinais em paralelo, ordenados em uma sequência padronizada. Uma UART pode realizar o inverso, ou seja, converte esses sinais sequenciados em sinais paralelos.

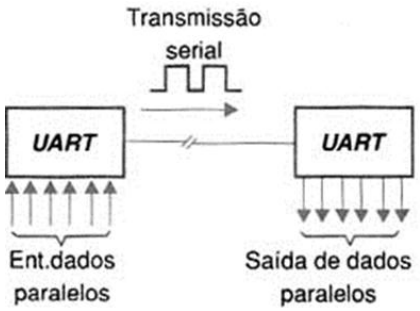

Figura 52: Funcionamento do UART

Existem vários tipos de UART e a diferença entre elas está na velocidade em que realizam a operação de conversão.

C.25 *Wiring*

Wiring é uma plataforma de prototipagem eletrônica de hardware livre composta por uma linguagem de programação, um ambiente de desenvolvimento integrado (IDE) e uma placa com microcontrolador. O sistema foi criado junto a designers e artistas de forma que usuários avançados, intermediários e iniciantes ao redor do mundo pudessem compartilhar suas ideias, conhecimentos e experiências coletivamente.

Wiring permite escrever programas para controlar aparelhos conectados a ele e assim criar todo o tipo de objetos interativos, correspondendo a experiência do usuário através do mundo físico. Com poucas linhas de código, por exemplo, é possível conectar- se a alguns componentes eletrônicos e observar a intensidade de uma luz variando conforme a distância que alguém chega a ela.

O projeto foi iniciado em 2003 por Hernando Barragán através do Interaction Design Institute Ivrea, Itália. Atualmente se desenvolve na Escola de Arquitetura e Design da Universidade de Los Andes, em Bogotá, Colômbia. Construído sobre o Processing, um projeto aberto de Casey Reas e Benjamin Fry, sua linguagem foi deselvolvida com a ajuda do Grupo de Computação e Estética da MIT Media Lab.