FBS

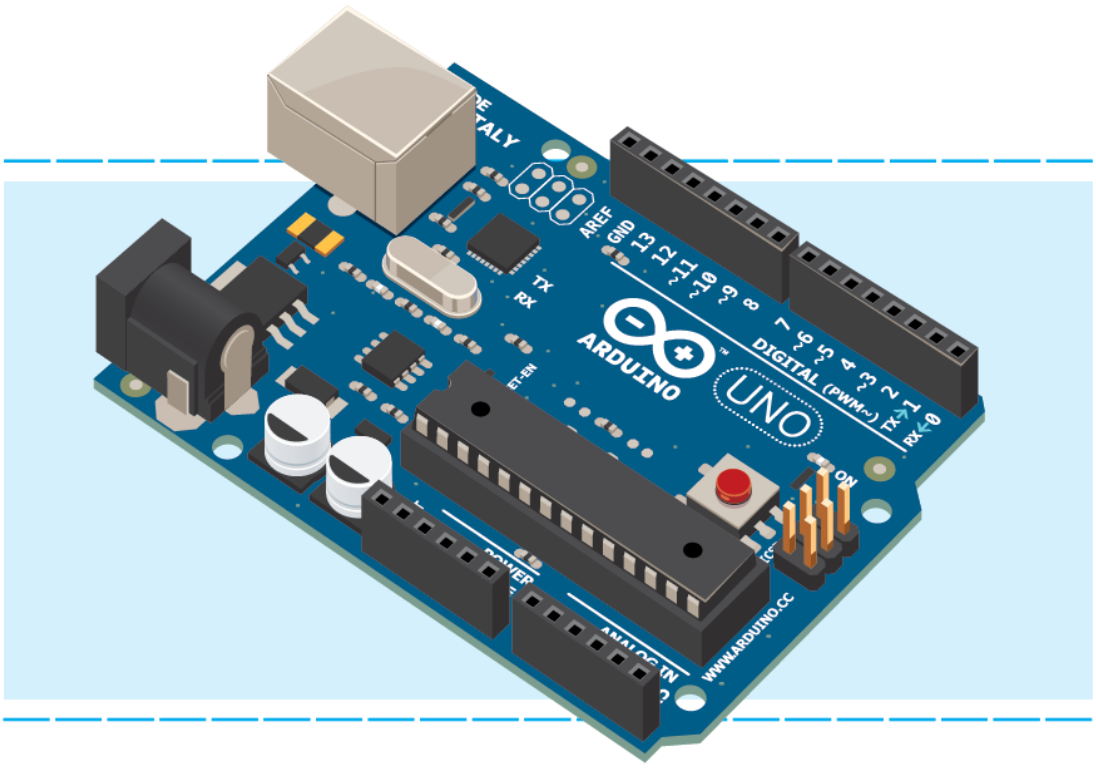
Eletrônica

APOSTILA ARDUINO

Com aplicações baseada na placa:

ARDUINO UNO

www.fbseletronica.com.br



V0RV1

Sumário

1 SOBRE A FBS ELETRÔNICA

A FBS Eletrônica é uma empresa que tem o objetivo de dar suporte para quem está começando na área de eletrônica.

Acesse nossos contatos na Internet:

Blog: www.fbseletronica.wordpress.com

Loja: www.fbseletronica.com.br

Facebook: www.facebook.com/fbseletronica

2 TERMOS DE USO

E-mail do desenvolvedor: fbseletronica@hotmail.com

3 OBJETIVOS

O objetivo desta apostila é apresentar a plataforma ARDUINO. Será baseada na plataforma ARDUINO UNO onde serão abordados os conceitos básicos de hardware e software com exemplos práticos.

4 INTRODUÇÃO

A plataforma Arduino está presente em muitos artigos na internet, porém para quem está iniciando na programação dessa plataforma geralmente sente dificuldade de um ter um ponto de partida. A fonte de principal de pesquisa deve ser o site do próprio Arduino, lá estão as informações básicas para iniciar. Um fator que bloqueia os iniciantes é o fato desse site ser em inglês. Essa apostila usa como referência principal o site do Arduino, e o objetivo é apresentar para os iniciantes um ponto de partida para essa plataforma. È abordado a teoria básica da plataforma, assim como a linguagem de programação e são apresentados no capítulo final alguns exemplos para colocar em prática.

Esperamos que essa apostila colabore com a comunidade que a cada dia que passa cresce mais.

Bons estudos!!

Caso encontrem erros (terão muitos), por gentileza nos enviem um e-mail para correção.

e-mail: fbseletronica@hotmail.com

5 O QUE É ARDUINO?

"ARDUINO é uma plataforma flexível open-source de hardware e software para prototipagem eletrônica. É destinada a designers, hobbistas, técnicos, engenheiros e pessoas interessadas em criar projetos ou ambientes interativos.

ARDUINO pode receber entradas de uma variedade de sensores e pode atuar o ambiente ao seu redor por controladores de iluminação, motores e uma infinidade de atuadores. O microcontrolador presente na placa é programado usando a linguagem de programação do ARDUINO (baseada em wiring) e o ambiente de desenvolvimento ARDUINO (baseada em processing). Os projetos com ARDUINO podem ser STAND-ALONE ou comunicar com um software rodando no PC. Os kits de desenvolvimento podem ser montados pelo próprio projetista ou podem ser compradas e o software pode ser baixado gratuitamente no site do ARDUINO. Os projetos de hardware para referências estão disponíveis em www.arduino.cc, sob licença open-source, onde está livre para adaptação as suas necessidades."

O ARDUINO é diferente das outras plataformas presentes no mercado devido aos seguintes fatores:

- É um ambiente multiplataforma, podendo ser executado em Windows, Macintosh e Linux;
- Tem por base um ambiente de fácil utilização baseado em processing;
- Pode ser programado utilizando um cabo de comunicação USB onde geralmente não é necessária uma fonte de alimentação;
- Possui hardware e software open-source, facilitando a montagem do seu próprio hardware sem precisar pagar nada aos criadores originais;
- Hardware de baixo custo;
- Grande comunidade ativa de usuários;
- Ambiente educacional, ideal para iniciantes que desejam resultados rápidos.

6 PLATAFORMA DE DESENVOLVIMENTO ARDUINO

O ARDUINO como foi visto anteriormente e formado por dois componentes principais: Hardware e software. O hardware é composto por uma placa de prototipagem na qual são construídos os projetos. O software é uma IDE, que é executado em um computador onde é feita a programação, conhecida como sketch, na qual será feita upload para a placa de prototipagem ARDUINO, através de uma comunicação serial. O sketch feito pelo projetista dirá à placa o que deve ser executado durante o seu funcionamento.

6.1 Hardware do ARDUINO

A placa do ARDUINO é um pequeno circuito microcontrolado, onde é colocado todos os componentes necessários para que este funcione e se comunique com o computador. Existem diversas versões de placas que são mantidas sob licença open-source. Nesta apostila será utilizada a placa ARDUINO UNO, porém as instruções aprendidas aqui podem ser aplicadas a outras placas. Em seguida é apresentada a placa ARDUINO UNO:

Conforme visto na imagem acima a placa Arduino UNO possui diversos conectores que servem para interface com o mundo externo. A seguir e dada uma explicação de como cada pino da placa pode ser utilizado.

- 14 pinos de entra e saída digital (pinos 0-13):

- Esses pinos podem ser utilizados como entradas ou saídas digitais de acordo com a necessidade do projeto e conforme foi definido no sketch criado na IDE.

- 6 pinos de entradas analógicas (pinos 0 - 5):
  - Esses pinos são dedicados a receber valores analógicos, por exemplo, a tensão de um sensor. O valor a ser lido deve estar na faixa de 0 a 5 V onde serão convertidos para valores entre 0 e 1023.
- 6 pinos de saídas analógicas (pinos 3, 5, 6, 9, 10 e 11):
  - São pinos digitais que podem ser programados para ser utilizados como saídas analógicas, utilizando modulação PWM.

A alimentação da placa pode ser feita a partir da porta USB do computador ou através de um adaptador AC. Para o adaptador AC recomenda-se uma tensão de 9 volts com pino redondo de 2,1 mm e centro positivo.

6.2 Software do ARDUINO

O software para programação do ARDUINO é uma IDE que permite a criação de Sketches para a placa ARDUINO. A linguagem de programação é modelada a partir da linguagem processing (www.processing.org). Quando pressionado o botão upload da IDE, o código escrito é traduzido para a linguagem C e é transmitido para o compilador avr-gcc, que realiza a tradução dos comandos para uma linguagem que pode ser compreendida pelo microcontrolador.

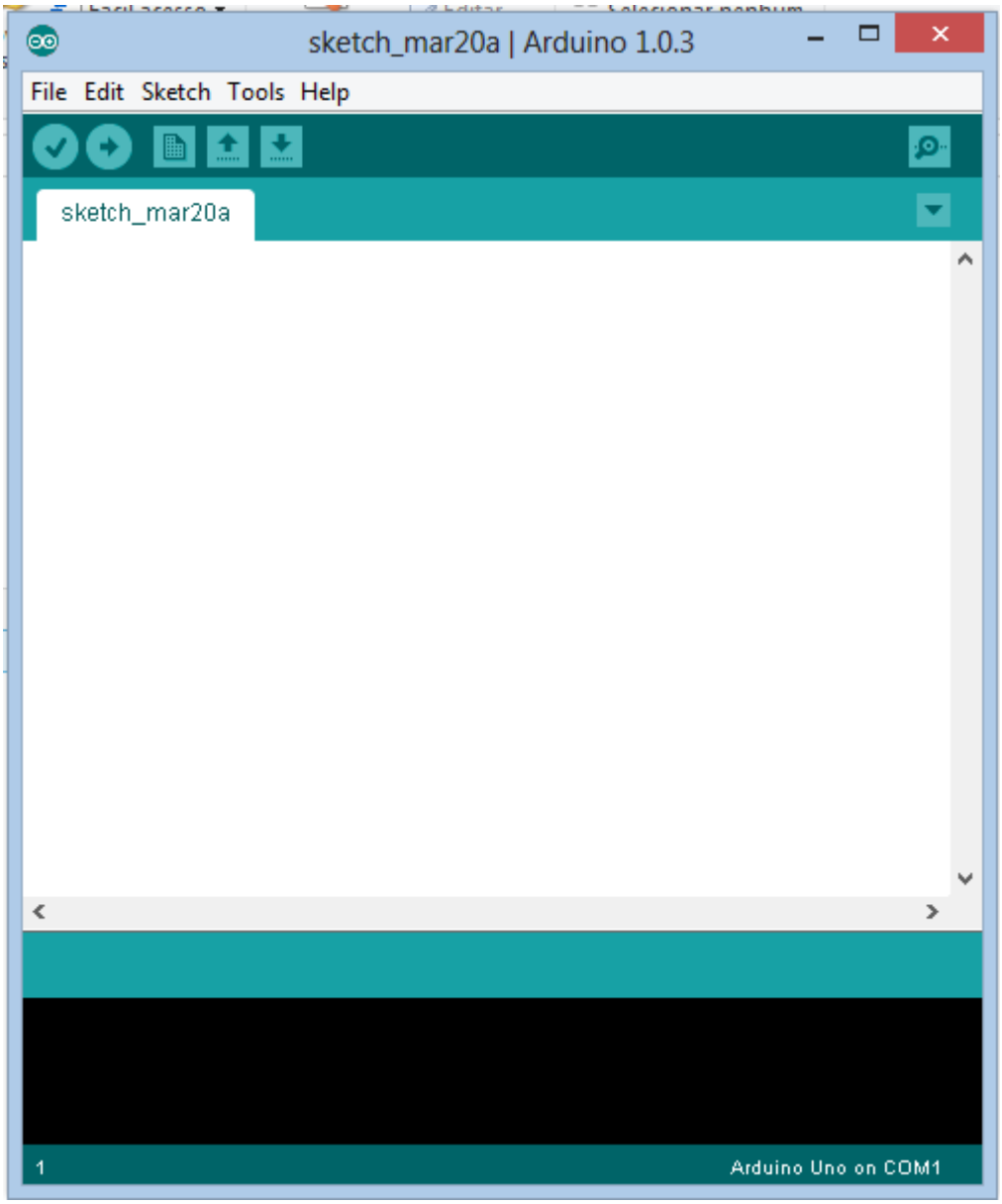
O Ciclo de programação do ARDUINO pode ser dividido da seguinte maneira:

1. Conexão da placa a uma porta USB do computador;
2. Desenvolvimento de um sketch com comando para a placa;
3. Upload do sketch para a placa, utilizando a comunicação USB.
4. Aguardar a reinicialização, após ocorrerá à execução do sketch criado.

A partir do momento que o software é gravado no ARDUINO não precisa mais do computador: o ARDUINO funciona como um computador independente e conseguirá sozinho executar o sketch criado, desde que seja ligado a uma fonte de energia.

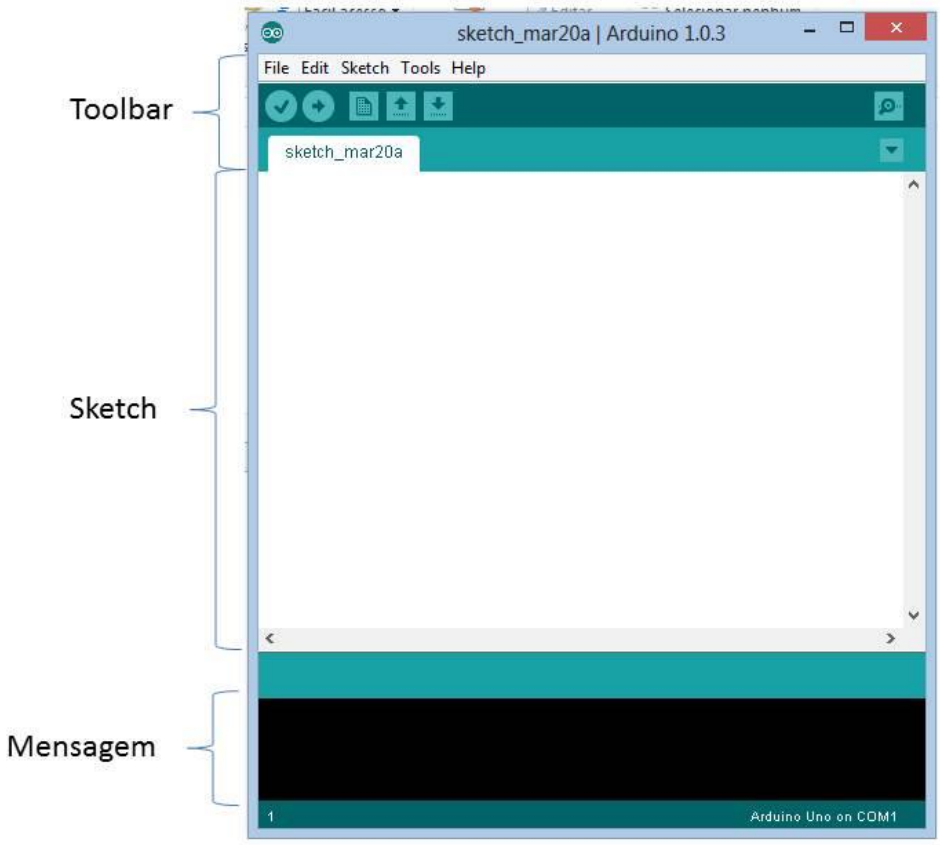
6.2.1 IDE do ARDUINO

Quando se abre o IDE do ARDUINO, será exibido algo semelhante à figura abaixo:



OBS.: Caso o sistema operacional utilizado seja diferente do Windows, pode haver algumas diferenças, mas o IDE é basicamente o mesmo.

O IDE é dividido em três partes: A Toolbar no topo, o código ou a Sketch Window no centro, e a janela de mensagens na base, conforme é exibido na figura abaixo:



Sob a Toolbar há uma guia, ou um conjunto de guias, com o nome do arquivo do sketch. Também há um botão posicionado no lado direito que habilita o serial monitor. Ao longo do topo há uma barra de menus, com os itens File, Edit, Sketch, Tools e Help. Os botões na Toolbar fornecem acesso conveniente às funções mais utilizadas dentro desses menus.

Abaixo são identificados os ícones de atalho da IDE:

- **Verify**
  - Verifica se existe erro no código digitado.
- **Upload**
  - Compila o código e grava na placa Arduino se corretamente conectada;
- **New**
  - Cria um novo sketch em branco.
- **Open**

  - Abre um sketch, presente no sketchbook.
- **Save**
  - Salva o sketch ativo
- **Seria monitor**
  - Abre o monitor serial.

Os demais comandos presentes na barra de menus podem ser consultados através do help do IDE.

7 PROGRAMAÇÃO

7.1 Estrutura de um programa ARDUINO

A estrutura básica de um programa para ARDUINO é extremamente simples e é dividida em dois blocos de funções. Primeiro bloco de função é chamado de setup () e o segundo é chamado de loop (). A estrutura é exibida a seguir:

```
void setup()

{

procedimentos;

}

void loop()

{

procedimentos;

}
```

A função setup() é responsável pela configuração inicial do ARDUINO e a função loop() é responsável pela execução das tarefas. Ambas as funções são requeridas para o correto funcionamento do ARDUINO.

A função setup segue logo abaixo da declaração de variáveis no inicio do programa. Esta é a primeira função a ser executada e é usada para configuração dos pinos ou inicialização da comunicação serial.

A função loop vem em seguida e inclui os comandos que serão executados durante o funcionamento do ARDUINO, por exemplo: leitura de entradas, acionamento de saídas, etc. Essa é a função principal do ARDUINO onde é executada a maior parte dos comandos.

7.2 Funções

Função é um bloco de código que possui um nome e procedimentos que são executados quando a mesma é chamada. As funções setup() e loop() foram explicados anteriormente e as demais funções do compilador serão apresentadas mais a frente neste capítulo.

Funções customizadas podem ser escritas para simplificar em tarefas repetitivas reduzindo repetições na programação. Funções são declaradas primeiramente definindo o seu tipo, que dirá qual o tipo de dado retornado pela função. Depois de definido o tipo de retorno

deve dar um nome a função e definir entre parêntese se algum parâmetro deve ser passado para a função, conforme exemplo de código a seguir:

```
tipo NomedaFunção(parametros)
{
Comandos;
}
```

A seguinte função do tipo inteiro chamada delayVal() é usada para atribuir um valor no programa através da leitura de um potenciômetro. Dentro do bloco da função primeiramente é declarado a variável local do tipo inteira chamada v, depois é atribuído o valor lido do potenciômetro pela função analogRead(pot) que retorna um valor entre 0 e 1023, depois este valor é divido por 4 para dar um resultado final entre 0 e 255. Finalmente é retornado este valor para o programa principal.

```
int delayVal()
{
int v;                    // create temporary variable 'v'
v = analogRead(pot);            // read potentiometer value
v /= 4;                   // converts 0-1023 to 0-255
return v;                 // return final value
}
```

Os seguintes símbolos são utilizados na construção de funções:

{ } – as chaves são utilizadas para delimitar os blocos de funções, dentro das chaves se encontram os procedimentos da função:

```
tipo função ()
{
procedimentos;
}
```

; - ponto e vírgula são utilizados para marcar o final do procedimento;

```
int x = 13; // declara a variável 'x' como um inteiro igual a 13
```

//- qualquer caractere despois das duas barras é ignorado pelo compilador, utilizado para comentários de apenas uma única linha.

```
// comentário de linha simples como no exemplo acima
```

`/*.....*/` – qualquer texto entre esses símbolos é ignorado pelo compilador, usado para comentar varias linhas.

```
/* comentário de múltiplas
Linhas geralmente utilizado
Para documentação ou para
Ignorar uma sequência de
Códigos*/
```

7.3 Constantes

A linguagem Arduino possui algumas constantes pré-definidas que auxiliam na programação. São divididas em grupos:

7.3.1 Definindo estados lógicos, true(verdadeiro) e false(falso)

Existem duas constantes para determinar valores booleanos na linguagem ARDUINO: true e false.

- **false**

Define um valor falso, ou seja, o valor booleano 0.

- **true**

Define valores verdadeiros, ou seja, o valor booleano 1.

7.3.2 Definindo Nível lógico alto e baixo nos pinos

Quando se trabalha com pinos de entrada ou saída digital apenas dois valores são possíveis: HIGH(alto, 1, 5 Volts) e LOW(baixo, 0, 0 Volts).

7.3.3 Definindo direção de um pino digital e Pull-up

Pinos digitais podem ser usados como INPUT_PULLUP INPUT, ou OUTPUT. Mudando um pino com pinMode () muda o comportamento elétrico do pino.

Para configura um pino como entrada utiliza a constante INPUT com a função pinMode().

Pode-se também colocar o pino como entrada e habilitar o resistor interno de o pull-up, dessa forma o pino passa a ser entrada e tem nível lógico 1 quando estiver aberto. Utiliza-se a constante INPUT_PULLUP na função pinMode().

Para configurar um pino como saída digital utiliza-se a constante OUTPUT na função pinMode().

7.4 Variáveis

Variáveis são posições na memória de programa do ARDUINO marcadas com um nome e o tipo de dado que irão ser armazenados nessa posição. Essa posição de memória pode receber valores durante a execução do programa e podem ser alterados a qualquer momento pelo programa e deve ser respeitado o tipo de dado da variável.

Ates de ser utilizada uma variável ela deve ser declara com um tipo e nome, e opcionalmente pode ser atribuir um valor a ela. E seguida é apresentado um trecho de código onde é declarada uma variável do tipo inteira chamada entradaAnalog e atribuído incialmente o valor zero e em seguida é atribuído o valor da entrada analógica 2 a esta variável, vejamos:

```
Int entradaAnalog = 0;                  //declara a variável e atribui o valor 0

entradaAnalog = analogRead(2);          //atribui o valor da entrada analógica 2
```

OBS.: Os nomes dados as variáveis podem descrever sua aplicação, para ficar mais fácil o entendimento do código durante o desenvolvimento ou em uma manutenção do mesmo no futuro.

7.5 Escopo de variáveis

Variáveis podem ser declaradas no inicio do programa antes da função setup() , dentro de funções e algumas vezes dentro de blocos de controles. O local onde uma variável é declarada determina o escopo da variável, ou seja o local onde tal variável pode ser utilizada.

Uma variável global de ser utilizado por qualquer função no programa, este tipo de variável é declarada fora de qualquer função no inicio do programa ates da função setup().

Uma variável local é declarada dentre de um bloco de função ou estrutura de controle. Ela somente poderá ser utilizada dentro deste bloco.

O seguinte exemplo mostra como declarar alguns tipos diferentes de variáveis em diferentes locais do programa:

```
int valor;  // 'valor' é uma variável global

void setup()
{

}
void loop()
```

```
{
for (int i=0; i<20;)    // 'i' é uma variável local
{                       //visível apenas no loop for
   i++;
}
float f;                // 'f' é uma variável local dentro de loop
}
```

7.6 Qualificadores de variáveis

7.6.1 static

static é utilizado para criar variáveis que são visíveis apenas em uma função. No entanto, ao contrario de variáveis locais que são criadas e destruídas a cada vez que uma função é chamada as variáveis estáticas mantém oi seu valor entre as chamadas de funções.

```
static tipo_dado nomedaVariável;
```

Exemplo:

```
static int valor;
```

7.6.2 volatile

É um qualificador de variável que é usado antes do tipo de dado para modificar o modo pelo qual o compilador e programa subsequente trata essa variável.

Especificamente, ele orienta o compilador a carregar a variável a partir da RAM e não a partir de um registador de armazenamento, o que é uma localização de memória temporária em que as variáveis do programa são armazenadas e manipuladas. Sob certas condições o valor de uma variável armazenada em registros pode ser imprecisa.

Uma variável deve ser declarada volátil sempre que o seu valor pode ser alterado por algo além do controle da seção de código em que ela aparece como na carente thread executada. No Arduino, o único lugar que isto é provável de ocorrer é em seções de código associado com interrupções, chamado de rotina de serviço de interrupção.

Exemplo:

```
// Inverte estado do LED quando interrupção é executada

int pin = 13;
volatile int state = LOW;

void setup()
{
```

```
  pinMode(pin, OUTPUT);
  attachInterrupt(0, blink, CHANGE);
}

void loop()
{
  digitalWrite(pin, state);
}
void blink()
{
  state = !state;
}
```

7.6.3 const

É um qualificador que modifica o comportamento da variável, tornando uma variável "read-only". Isto significa que a variável pode ser utilizada tal como qualquer outra variável do seu tipo, mas o seu valor não pode ser alterado. Você receberá um erro do compilador se você tentar atribuir um valor a uma variável const.

Constantes definidas com a palavra-chave const obedecer as regras de escopo de variáveis que governam outras variáveis.

Exemplo:

```
const float pi = 3.14;
float x;

// ....

x = pi * 2;   // é legal usar const em expressões matemáticas

pi = 7;      // illegal - you can't write to (modify) a constant
```

Você pode usar const ou **#define** para criar constates numéricas ou constantes de textos. Para vetores você vai precisar usar const. Em geral é preferível a **const** a **#define** para definir constantes.

7.7 Tipos de dados

As variáveis podem assumir diferentes tipos de dados, tais tipos determinarão sua capacidade e numeração que poderá ser utilizada. Os tipos básicos de dados são apresentados a seguir:

7.7.1 void

A palavra reservada **void** é usada em declarações de funções. Este tipo indica que a função não retorna nenhum valor quando é executada.

Exemplo:

```
// as funções setup e loop não retornam valores quando são executadas
void setup()
{
  // ...
}


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

7.7.2 Boolean

O tipo **boolean** pode representar valores booleanos, verdadeiro (true) ou falso(false). Um tipo **boolean** ocupa um byte da memória.

Exemplo:

```
int LEDpin = 5;        // LED no pino 5
int switchPin = 13;    // chave no 13

boolean running = false;           //variável booleana

void setup()
{
  pinMode(LEDpin, OUTPUT);         //configura pino como saída
  pinMode(switchPin, INPUT);       //configura pino como entrada
  digitalWrite(switchPin, HIGH);      //Liga pullup do pino
}

void loop()
{
  if (digitalRead(switchPin) == LOW)  //se pino em nível baixo
  {                                   // switch is pressed -
                                      // pullup keeps pin high
                                      //normally

    delay(100);                          // delay to debounce switch
    running = !running;                  // toggle running variable
    digitalWrite(LEDpin, running)        // indicate via LED
  }
}
```

7.7.3 char

O tipo char armazena valores de 1 byte. Caracteres são codificados em um único byte e são especificados na tabela ASCII. O tipo char é sinalizado e representa números de -128 a 127. Para números não sinalizados usa-se o tipo byte.

Exemplo

```
char myChar = 'A';
char myChar = 65;       // both are equivalent
```

7.7.4 unsigned char

unsigned char armazena valores de 1 byte não sinalizados, é mesmo que utiliza o tipo byte. A faixa de valores vai de 0 a 255.

Para programação consistente no estilo Arduino o tipo byte é preferido para esse tipo de dado.

Exemplo

```
unsigned char myChar = 240;
```

7.7.5 byte

Armazena valores de 8 bits não sinalizados de 0 a 255.

Exemplo

```
byte b = B10010;  // B10010 = 18 decimal
```

7.7.6 int

Inteiros são tipos primários de armazenamento. No Arduino Uno (e em outras placas baseadas em ATMEGA) um int armazena valores de 16 bits(2 bytes). Esse tipo compreende valores de -32768 a 32767. Já no Arduino Due, um int armazena valores de 32 bits (4 bytes) que compreende valores de -2147483648 a 2,147483647.

Exemplo:

```
int ledPin = 13;
```

Quando o valor contido na variável excede o seu valor máximo o seu valor é reiniciado para o mínimo. Exemplo

```
int x;
x = -32768;         //atribui o valor mínimo a variável x
x = x - 1;        // x agora contém o valor máximo: 32,767
```

```
x = 32767;
x = x + 1;       // x agora contém o valor mínimo: -32,768
```

7.7.7 unsigned int

No Arduino UNO e em outras placas baseadas em ATMEGA armazenam valores de 16 bits não sinalizados, ou seja, apenas valores negativos de 0 a 65535.

O Arduino DUE armazena valores de 32 bits não sinalizados, e compreende a faixa de 0 a 4294967295.

Exemplo

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

7.7.8 word

O tipo armazena valores de 16 bis não sinalizados que compreendem valores de 0 a 65535.

Exemplo

```
word w = 10000;
```

7.7.9 long

O tipo de dado Long armazena valores inteiros sinalizados de 32 bits (4 bytes) que compreendem a faixa de -2147483648 a 2147483647

Exemplo

```
long speedOfLight = 186000L;
```

7.7.10 unsigned long

O tipo unsigned long armazena valores de 32 bits(4 bytes) não sinalizados que compreendem a faixa de 0 a 429967295.

Exemplo

```
unsigned long time;

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

void loop()
```

```
{
  Serial.print("Time: ");
  time = millis();

  Serial.println(time);

  delay(1000);
}
```

7.7.11 short

O tipo short armazena valores de 16 bits (2 bytes) sinalizados.

Exemplo

```
short ledPin = 13;
```

7.7.12 float

O tipo float armazena valor em ponto flutuante, ou seja, um valor que possui casas decimais. O tipo float armazena valores de 32 bits (4 bytes) e compreendem a faixa de -3,4028235 E+38 a 3,4028235 E+38.4.

A matemática em ponto flutuante requer muito processamento, por exemplo, se for feita uma operação em ponto flutuante dentro de um loop, ocorrerá um atraso maior, do que se fosse feita uma operação com inteiros. Deve-se ficar atento ao uso do tipo float tanto na questão de espaço de memória quanto no processamento. As funções de manipulação do tipo float requerem muito processamento,

Exemplos

```
float myfloat;
float sensorCalbrate = 1.117;
```

7.7.13 double

O tipo double também armazena valores de ponto flutuante, porém no Arduino Uno e outras placas baseadas em ATMEGA esse tipo é exatamente o mesmo que o tipo float, sem ganho de precisão, já no Arduino Due o tipo double pussui 64 bits (8 bytes) provendo maior precisão, seu valor máximo pode chegar a 1,7976931348623157 x 10^{308}.

7.8 Vetores

Uma matriz é uma coleção de variáveis que são acessadas por meio de um índice. Matrizes em um programa em linguagem C, na qual o Arduino é baseado, pode ser um pouco

complicado. Mas utilizando uma matriz de linha simples, conhecida como vetores, pode tornar a programação mais simples.

7.8.1 Criando e declarando um vetor

Todos os métodos abaixo são válidos para criar(declarar) um vetor:

- Pode-se declara um vetor sem inicializa-lo:

  int myInts[6];

- Pode-se inicializar um vetor sem determinar o tamanho, apenas inicializando com os elementos. O compilador se encarrega de determinar o tamanho conforme a quantidade de elementos:

```
int myPins[] = {2, 4, 8, 3, 6};
```

- Pode-se inicializar o vetor com tamanho e com as variáveis:

```
int mySensVals[6] = {2, 4, -8, 3, 2};
char message[6] = "hello";
```

7.9 string (vetor do tipo char)

É um conjunto de caracteres ASCII utilizado para armazenar informações textuais (uma string pode ser utilizada para enviar dados pela serial, ou para exibir dados em um display LCD). Uma String utiliza um byte para cada caractere e mais um caractere nulo para indicar o fim da string.

Possibilidade para declaração de strings:

- Declarar um vetor de char sem inicialização:

  ```
  char Str1[15];
  ```

- Declarar um vetor de char com um caractere adicional que o compilador encarregará de colocar o caractere nulo:

  ```
  char Str2[8] = {'a', 'r', 'd', 'u', 'i', 'n', 'o'};
  ```

- Declarar um vetor de char inicializando com todos os caracteres inclusive o caractere nulo

  ```
  char Str3[8] = {'a', 'r', 'd', 'u', 'i', 'n', 'o', '\0'};
  ```

- Inicializar o vetor sem tamanho determinado e atribuir uma string.O compilador se encarrega de determinar o tamanho, conforme tamanho da string atribuída e adiciona o caractere nulo automaticamente:

  ```
  char Str4[ ] = "arduino";
  ```

- Inicializar vetor com tamanho determinado e atribuir uma string para inicializar. Importante observar que o tamanho deve ser suficiente para conter o caractere nulo:

```
char Str5[8] = "arduino";
```

- Inicializar o vetor com tamanho extra e inicializar com uma string de quantidades de caracteres menor que a quantidades de variáveis no vetor:

```
char Str6[15] = "arduino";
```

7.10 Vetores de string

Muitas vezes, é conveniente, quando se trabalha com grandes quantidades de texto, configurar um vetor de string para manipulação deste texto.

No código em seguida o caractere asterisco após o tipo char(char*) indicar que se trata de um vetor de string. Todos os nomes nos vetor são na verdade ponteiros para outros vetores. Isso é necessário para criar vetores de vetores:

Exemplo:

```
char* myStrings[]={"This is string 1", "This is string 2", "This is string 3","This is string 4", "This is string 5","This is string 6"};

void setup(){

Serial.begin(9600);

}

void loop(){

for (int i = 0; i < 6; i++){

   Serial.println(myStrings[i]);

   delay(500);

   }

}
```

7.11 Operadores

A linguagem C é muito rica em operadores sendo possivelmente uma das linguagens com maior numero de operadores disponíveis. Possui operadores encontrados em linguagens de alto nível como operadores encontrados em linguagens de baixo nível como o Assembly.

Os operadores são divididos nas seguintes categorias: Atribuição, aritméticos, relacionais, lógicos, lógicos bit a bit.

7.11.1 Atribuição

O operador de atribuição em linguagem C é o "=", que é utilizado para atribuir determinado valor a uma variável. Exemplo:

```
x = 10;
y = x+3;
```

Verificando as linhas anteriores duas operações de atribuição foram feitas. Primeiro foi atribuído o valor 10 a variável "x", e na segunda foi atribuído o resultado da soma entre a variável x(que é 10) com o valor 3, portanto ao final desta operação será atribuído o valor 13 a variável "y". A atribuição é sempre feita da direita para a esquerda.

7.11.2 Aritméticos

São utilizados para efetuar operações matemáticas, a tabela abaixo exibe os operadores aritméticos disponíveis na linguagem C:

OPERADOR	AÇÃO
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
%	Resto de divisão inteira
++	Incremento
--	Decremento

Os operadores de **adição**, **subtração**, **multiplicação** e **divisão** possuem a mesmas funções já conhecida em matemática.

O operador **%** é utilizado para retornar o resto de uma divisão inteira. Exemplo:

```
x = 5%2; //retorna para a variável x o valor 1 que é o resto de uma divisão inteira
de 5 por 2.
```

Os operadores de **incremento** e **decremento** são utilizados para adicionar ou subtrair em 1 o valor de uma variável.

Exemplos:

```
x= 10;
x++; //incrementa em 1 o valor da variável x, portanto x valerá 11
x = 20;
x--; //decrementa em 1 do valar da variável x, portanto x valerá 19
```

7.11.3 Operadores Relacionais

São utilizados em testes condicionais para determinar a relação entre dados, são os seguintes:

OPERADOR	AÇÃO
>	Maior que
>=	Maior ou igual que
<	Menor que
<=	Menor ou igual que
==	Igual a
!=	Diferente de

O funcionamento destes operadores é idêntico ao estudado em matemática. Eles retornam **verdadeiro** ou **falso** conforme a condição testada. São bastante utilizados em estruturas de controle onde a partir de teste o programa toma caminhos diferentes, como será visto mais a frente. Exemplo:

```
a = 5;
b = 10;
x = a > b;   //neste caso x valerá 0, pois 5 é menor do que 10.
```

É importante observar que o operador relacional de igualdade lembra bastante o operador de atribuição. Um erro muito comum para iniciantes em linguagem C é o uso do operador de atribuição em testes relacionais. Deve-se ficar atento, pois usar o operador de atribuição ao invés do relacional de igualdade pode ocasionar erros de lógica no programa.

7.11.4 Operadores Lógicos

Os operadores lógicos são utilizados para realizar testes booleanos entre elementos em um teste condicional. Eles retornam **verdadeiro** ou **falso** conforme o resultado do teste. São os seguintes:

OPERADOR	AÇÃO
&&	AND(E)
||	OR(OU)
!	NOT(NÃO)

Os operadores lógicos são muito importantes na construção de testes condicionais, com eles pode-se relacionar diversas condições em um mesmo teste. Exemplo:

```
x = 10;
y = x>>5 && x<<20;    //neste caso y valerá 1, pois 10
```

```
//é maior que 5 E menor do que 20
```

Esses operadores são muito utilizados em conjuntos com os operadores relacionais em estruturas de controle.

7.11.5 Operadores Lógicos Bit a Bit

Os operadores Lógicos bit a bit são utilizados para operações lógicas entre elementos ou variáveis. São os seguintes:

OPERADOR	AÇÃO
&	AND(E)
|	OR(OU)
^	XOR (OU EXCLUSIVO)
~	NOT (NÃO)
>>	Deslocamento à direita
<<	Deslocamento à esquerda

O operador **& (AND)** faz a operação booleana **AND** para cada bit dos operandos.
Exemplo:

```
int v1, v2;
v1 = 0x5A;
v2 = v1 & 0x0F;
```

A operação ocorrerá da seguinte forma:

```
0x5A hexadecimal  = 0 1 0 1 1 0 1 0 binário
                    AND (&)
0x0F Hexadecimal  = 0 0 0 0 1 1 1 1 binário
       Resultado  = 0 0 0 0 1 0 1 0 binário = 0x0A hexadecimal
```

Portanto o valor armazenado em v2 será 0x0Ah ou 10d.

O operador **AND** é bastante utilizado para desligar bits de variáveis ou registradores.

O operador **|| (OR)** faz a operação booleana **OR** para cada bit dos operandos.

Exemplo:

```
int v1, v2;
v1 = 0x01;
v2 = v1 || 0xF0;
```

A operação ocorrerá da seguinte forma:

```
0x20 hexadecimal  = 0 0 0 0 0 0 0 1 binário
```

OR(||)

```
0x04 Hexadecimal  = 1 1 1 1 0 0 0 0 binário
        Resultado = 1 1 1 1 0 0 0 1 binário = 0xF1 hexadecimal
```

Portanto o valor armazenado em v2 será 0xF2h ou 242d.

O operador OR é muito utilizado para ligar bits de variáveis ou registradores.

O operador **^(XOR)** faz a operação booleana **XOR** entre dois operandos, lembrando que a operação **XOR** resulta em 0 quando os valores são iguais e resulta em 1 quando os valores são diferentes.

```
int v1, v2;
v1 = 0x50;
v2 = v1 ^ 0x50;
```

A operação ocorrerá da seguinte forma:

```
0x20 hexadecimal  = 0 1 0 1 0 0 0 0 binário
```

XOR(^)

```
0x04 Hexadecimal  = 0 1 0 1 0 0 0 0 binário
        Resultado = 0 0 0 0 0 0 0 0  binário = 0x00hexadecimal
```

Portanto o valor armazenado em v2 será 0x00h ou 00d.

O operador **XOR** e muito utilizado para fazer a comparação entre variáveis e teste de registradores.

O operador **~(NOT)** faz a operação booleana **NOT**, ou complemento de uma variável. Exemplo:

```
int  x , y;
x = 0b00001111;
y = ~x;
```

O resultado desta operação será:

y = 0b11110000;

O operador **NOT** é muito utilizado para inverter estados de bit em variáveis, registradores e pinos de I/O.

Os operadores de deslocamento são utilizados para deslocar determinadas quantidades de bits para direita ou esquerda. A forma geral para o uso destes operadores é:

Valor **>>** quantidade de bits a ser deslocados para a direita;

Valor **<<** quantidade de bits a ser deslocados para a esquerda.

Exemplo:

```
int x, y, z;
x = 10;
y = x<<2;
z = x>>1;
```

As operações acima funcionam da seguinte forma:

Primeiramente foi atribuído a variável y o valor da variável x com deslocamento de dois bits à esquerda:

```
10 decimal = 00001010
```

Deslocando 2 bits para a esquerda resultará em: 00101000 = 40 decimal.

Depois foi atribuído a variável z o valor da variável x deslocado em um bit para a direita, que resultou em: 00000101 = 5 decimal.

Nota-se que ao deslocar em um bit para a esquerda na verdade a variável está sendo multiplicada por dois e ao deslocar um bit para a direita a variável está sendo dividida por 2.

7.12Associação de operadores

Na linguagem C há a possibilidade de abreviação de operadores em atribuições. Uma operação de atribuição possui geralmente a seguinte forma:

variável = variável (operando) valor ou variável

Com a abreviação de operadores o comando ficara reduzido da seguinte forma:

variável (operando) = valor ou variável

A seguir é exibido os tipos de operadores abreviados em C:

Forma expandida	**Forma reduzida**
x = x+ y	x += y
x = x- y	x -= y
x = x* y	x *= y
x = x / y	x /= y
x = x % y	x %= y
x = x & y	x &= y
x = x | y	x |= y
x = x^y	x ^= y

x = x<< y	x <<= y
x = x>>y	x >>= y

7.13 if

O comando **if** é utilizado para executar um comando ou bloco de comandos no caso de uma determinada condição ser avaliada como verdadeira.

A forma geral do comando **if** é:

```
if (condição)  comandoA;
```

Caso seja necessário executar um comando se a condição for avaliada com Falsa usa a clausula **else** junto com o comando **if**, o comando ficará da seguinte forma**:**

```
if (condição)
  comandoA;
else
 comandoB;
```

Desta forma é possível executar comandos conforme o resultado do teste feito.

Pode-se executar um bloco de código dentro da declaração **if – else**, pra isso deve-se limitar com "{" e "}", exemplo:

```
if(condição)
{
  comandoA1;   //bloco de código para a condição verdadeira
  comandoaA2;
  ...
 comandoAn;
}
else
{
  comandoB1;   //bloco de código para a condição falsa
  comandoB2;
  ...
  comandoBn;
 }
```

Deve-se ficar claro que a clausula **else** não é necessária para o funcionamento do comando **if**, funciona apena como uma opção para tomar uma ação se a condição testada for avaliada como falsa. Desta forma a maneira mais simples de seu utilizar o comando **if** é:

```
if(condição) comandoA;
```

Há a possibilidade de aninhar diversos comandos **if,** um dentro do outro para executar uma sequência de testes, exemplo:

```
if(condição1) comandoA;
    else if(condição2) comandoB;
```

```
        else if(condição3) comandoC;
                …
```

Neste caso será avaliada a **condição1**, se verdadeira será executado o **comandoA** e finalizada a estrutura de teste, caso a c**ondição1** seja falsa será avaliada a **condição2**, caso verdadeira será executado o **comandoB** e finalizado a estrutura, caso a **condição2** seja falsa será avaliada a **condição3** e assim por diante até finalizarem os testes.

Também é possível utilizar estruturas **if-else** dentro de estruturas **if-else, como** por exemplo**:**

```
 if(condição1)
 {
       if(condição2)
        {
          comandoA;
           comandoB;
        }
       else
       {
          comandoC;
           comandoD;
       }
}
else
{
       if(condição3)
        {
          comandoE;
           comandoF;
        }
       else
       {
          comandoG;
           comandoH;
       }
}
```

Neste caso a **condição1** é avaliada caso verdadeira será avaliada a **condição2.** Caso a **condição1** seja avaliada como falsa será avaliada a **condição3.**

7.14 switch

Em alguns caso, como na comparação de uma determinada variável com diversos valores diferentes o comando **if** pode torna-se extenso, confuso e pouco eficiente.

O comando **switch** permite a realização de comparações sucessivas de uma forma elegante, clara e eficiente. O formato do comando **switch** é:

```
switch(variável)
{
       case constante1:
```

```
            comandoA;
            . . .
      break;
         case constante2:
            comandoB;
            . . .
      break;
        . . .
        . . .
        default:
        comandoX;
}
```

O valor da variável é testado com os valores especificados pelas clausulas case. Caso a variável possua o mesmo valor de uma das constantes, os comandos seguintes àquela clausula case serão executados. Caso o valor da variável não for igual a nenhum valor das constantes especificadas então os comandos após à clausula default serão executados.

Em cada sequência de comandos da clausula case é encerrado por uma clausula **break**. Está clausula tem a função de interromper o a estrutura switch de realizar outros testes. Caso esta clausula seja omitida os comandos seguintes de outra clausula case serão executados até ser encontrada uma clausula **break** ou chegar ao fim do switch. Portanto deve-se ficar atento ao uso da clausula **break** para determinar o fim dos comandos de cada clausula case.

Algumas características importantes do comando **switch** devem ser observadas:

- O comando **switch** somente pode testar igualdades. Não são admitidos outros operadores como no comando **if.**
- Somente números inteiros e caracteres podem ser usados.
- Dentro de um mesmo **switch** não poderá haver cases com constantes iguais;
- As cláusulas **break** e a **default** são opcionais.

7.15 for

O comando **for** é uma das mais comuns estruturas de repetição utilizadas, sendo bem poderosa na linguagem C em relação a outras linguagens de programação.

A forma geral do for é:

```
for( inicialização;condição;incremento) comando;
```

ou

```
for( inicialização;condição;incremento)
{
     Comando1;
     Comando2;
     . . .
```

```
    ComandoN;
}
```

Cada uma das três seções do comando **for** possui uma função distinta conforme em seguida:

Inicialização: esta seção conterá uma expressão valida utilizada normalmente pra inicializar a variável de controle do laço.

Condição: esta seção pode conter a condição a ser avaliada para decisão de continuidade do laço de repetição. Enquanto a condição for avaliada como verdadeira o laço permanecerá em execução.

Incremento: esta seção pode conter uma declaração para incremento da variável de controle do laço.

O funcionamento básico do comando **for** é o seguinte:

Primeiramente a seção de inicialização é executada, em seguida a condição é testada e se for verdadeira o comando ou bloco de comandos são executados, após o fim dos comandos o laço executa a seção de incremento da variável de controle.

É possível utilizar a clausula **break** para encerrar a execução do laço for independente de a condição de repetição ser avaliada, desta forma quando o comando **break** for executado dentro de um laço de repetição **for,** este laço é interrompido imediatamente.

7.16 while

Outro tipo de estrutura de repetição da linguagem C é comando **while**, que é utilizado para repetir um comando ou um conjunto de instruções enquanto uma condição for avaliada como verdadeira. No comando **while** existe apenas o teste condicional, diferentemente do comando **for** que possui ciclos definidos. A estrutura do comando **while** é a seguinte:

```
while(condição)
{
    comando1;
    comando2;
    ...
    comandoN;
}
```

O funcionamento do comando **while** ocorre de uma forma bem simples. Primeiramente a condição é avaliada, caso a condição seja avaliada como verdadeira o comando ou o bloco de comandos é executado em seguida a condição é avaliada

novamente. Caso seja avaliada como verdadeira o laço é repetido, quando avaliada como falsa o laço é interrompido.

Assim como no comando **for** é possível usar a clausula **break** para interromper prematuramente um laço **while**.

7.17 do - while

O comando **do-while** é utilizado para criar uma estrutura de repetição com funcionamento ligeiramente diferente do **while** e **for** tradicionais.

De fato, a diferença entre a estrutura **while** e a estrutura **do-while** é que esta última realiza a avaliação da condição de teste no final de cada ciclo de iteração do laço de repetição, ao contrário do comando **while**, que realiza o teste no início de cada ciclo.

A forma geral da estrutura **do-while** é:

```
do
 comando;
while (condição);
```

Ou

```
do
{
   comandoA;
   comandoB;
...
   comandoN;
} while (condição);
```

O funcionamento da estrutura **do-while** é o seguinte: o comando ou bloco de comandos é executado e logo depois será avaliada a condição de teste, caso ela seja verdadeira, será iniciada uma nova iteração do ciclo, caso seja falsa, o laço é terminado.

Assim como nos comandos **for** e **while** pode-se utilizar a clausula **break** para interromper o laço **do-while**.

7.18 Diretivas do compilador

7.18.1 #define

É uma ferramenta que permite ao programador dar um nome a um valor constante antes de de iniciar a programação. O compilador substituirá esse a constante no lugar deste nome na hora da compilação. Isto facilita na programação e deixa o código mais agradável.

A sintaxe é mesma da linguagem C:

```
#define constantName valor
```

Exemplo:

```
#define ledPin 3
```

7.18.2 #include

È utilizado para incluir bibliotecas externas ao scketch. Dessa forma pode-se acessar um grande número de bibliotecas seja ela do padrão da linguagem C ou as feitas especialmente para Arduino, ou até mesmo bibliotecas feitas pelo programador.

Exemplo:

```
#include <biblioteca.h >
```

Obs.: Assim como a diretiva #define não se coloca ‘;’ no final.

7.19 Funções do ARDUINO

O Arduino já possui funções internas para manipulação de I/O digital e analógico assim como funções para manipulações de bits, matemáticas, comunicação entre outras. A seguir serão apresentadas algumas dessas funções.

7.19.1 Funções de Entrada/Saída Digital

pinMode()

Configura um pino especifico para ser entrada ou saída digital.

Sintaxe:

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

Parametros:

Pino: deve-se colocar o numero correspondente ao pino que se deseja configurar, conforme placa que está trabalhando.

Modo: deve-se colocar o modo que deseja configurar o pino. INPUT, OUTPUT, INPUT_PULLUP.

Exemplo:

```
int ledPin = 13;            // LED connected to digital pin 13

void setup()
{
  pinMode(ledPin, OUTPUT);    //sets the digital pin as output
}

void loop()
{
  digitalWrite(ledPin, HIGH);  //sets the LED on
  delay(1000);                  // waits for a second
  digitalWrite(ledPin, LOW);  // sets the LED off
  delay(1000);                // waits for a second
}
```

digitalWrite()

Coloca um nível lógico Alto (HIGH, 5V) ou baixo (LOW, 0V) em um pino configurado como saída digital.

Sintaxe

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

Parâmetros

pino: Numero correspondente ao pino;

valor: HIGH OU LOW

Exemplo

```
 int ledPin = 13;                 // LED connected to digital pin 13


void setup()
{
  pinMode(ledPin, OUTPUT);      // sets the digital pin as output
}

void loop()
{
  digitalWrite(ledPin, HIGH);   // sets the LED on
  delay(1000);                  // waits for a second
  digitalWrite(ledPin, LOW);    // sets the LED off
  delay(1000);                  // waits for a second
}
```

digitalRead()

Lê o valor presente e um pino digital. Este valor pode ser HIGH ou LOW.

Sintaxe:

```
digitalRead(pino)
```

Parâmetros:

Pino: valor correspondente ao pino que se deseja ler.

Retorno

HIGH ou LOW.

Exemplo:

int ledPin = 13; // LED connected to digital pin 13

```
int inPin = 7;    // pushbutton connected to digital pin 7
int val = 0;      // variable to store the read value

void setup()
{
  pinMode(ledPin, OUTPUT);      // sets the digital pin 13 as output
  pinMode(inPin, INPUT);      // sets the digital pin 7 as input
}
void loop()
{
  val = digitalRead(inPin);   // read the input pin
  digitalWrite(ledPin, val);  // sets the LED to the button's value
}
```

7.19.2 Funções de Entrada Analógica

`analogReference()`

Configura a referencia de tensão para a conversão analógica/digital.

Os tipos possíveis de configurações são:

- DEFAULT: a tensão padrão para conversão é a tensão de alimentação da placa. 5 Voltas para placas alimentadas com 5 volts e 3.3 Voltas para placas alimentadas com 3.3 V.
- INTERNAL1V1: referência de 1,1V, Apenas no Arduino mega;
- INTERNAL2V56: referência interna de 5,6 V, apenas no Arduino mega;
- EXTERNAL: Referência de tensão aplicada no pino AREF(valor entre 0 e 5V).

Sintaxe:

```
analogReference(tipo)
```

Parametros:

TIPO: DEFAULT, INTERNAL1V1, INTERNAL2V56, EXTERNAL.

`analogRead()`

Lê o valor presente em um pino configurado como entrada analógica. Internamente o Arduino possui um conversor A/D de 10 bits. Dessa forma o valor retornado por esta função estará na faixa de 0 a 1023 conforme o valor presente no pino.

Sintaxe:

```
analogRead(pino)
```

Parâmetros:

Pino: valor do pino configurado como entrada analógica(0 a 5 na maioria da placas, 0 a 7 na MINI e NANO, 0 a 15 na MEGA).

Retorno

Int(0 a 1023)

Exemplo:

```
int analogPin = 3;     // potentiometer wiper (middle terminal) connected to analog
pin 3
                       // outside leads to ground and +5V
int val = 0;           // variable to store the value read

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

void loop()
{
  val = analogRead(analogPin);    // read the input pin
  Serial.println(val);             // debug value
}
```

7.19.3 Funções de temporização

`millis()`

Retorna o numero de milissegundos desde que a placa começou a executar o programa. Este número retorna a zero após aproximadamente 50 dias.

Sintaxe:

```
time = millis();
```

Parâmetros:

Nenhum

Retorno:

```
unsigned long
```

Exemplo:

```
unsigned long time;

void setup(){
  Serial.begin(9600);
}
void loop(){
  Serial.print("Time: ");
  time = millis();
  //prints time since program started
  Serial.println(time);
  // wait a second so as not to send massive amounts of data
  delay(1000);
}
```

micros()

Retorna o tempo em microssegundos que a placa está executando o programa. Este numero retorna a zero em aproximadamente 70 minutos.

Sintaxe:

```
tempo = micros();
```

Parâmetros:

Nenhum

Retorno:

```
unsigned long
```

Exemplo:

```
unsigned long time;

void setup(){
  Serial.begin(9600);
}
void loop(){
  Serial.print("Time: ");
  time = micros();
  //prints time since program started
  Serial.println(time);
  // wait a second so as not to send massive amounts of data
  delay(1000);
}
```

delay()

Pausa o programa por um tempo em milissegundos passado no parâmetro.

Sintaxe:

```
delay(ms)
```

Parâmetros:

ms: valor de milissegundos que se deseja esperar(unsigned long)

Retorno:

Nenhum

Exemplo:

```
int ledPin = 13;                 // LED connected to digital pin 13

void setup()
{
  pinMode(ledPin, OUTPUT);      // sets the digital pin as output
}

void loop()
{
  digitalWrite(ledPin, HIGH);   // sets the LED on
  delay(1000);                  // waits for a second
  digitalWrite(ledPin, LOW);    // sets the LED off
  delay(1000);                  // waits for a second
}
```

delayMicroseconds()

Pausa o programa pelo tempo em microssegundos passado no parâmetro.

O valor máximo que produz um valor preciso é 16383.

Sintaxe:

delayMicroseconds(us)

Parâmetros:

us: valor em microssegundos que se deseja esperar(unsigned int)

Exemplo:

```
int outPin = 8;                 // digital pin 8

void setup()
{
  pinMode(outPin, OUTPUT);      // sets the digital pin as output
}

void loop()
{
  digitalWrite(outPin, HIGH);   // sets the pin on
  delayMicroseconds(50);        // pauses for 50 microseconds
  digitalWrite(outPin, LOW);    // sets the pin off
  delayMicroseconds(50);        // pauses for 50 microseconds
}
```

7.19.4 Funções de bit e byte

lowByte()

Retorna o byte menos significativos de uma variável

Sintaxe

lowByte(x)

Parâmetros

x: um valor de qualquer tipo

highByte()

Retorna o byte mais significativo de uma variável.

Sintaxe

highByte(x)

Parâmetros

x: um valor de qualquer tipo

bitRead()

Lê um bit de um numero

Sintaxe

```
bitRead(x, n)
```

Parâmetros

x: variável que deseja ser lido o bit

n: a posição do do bit na variável.

Retorno

O valor do bit (0 ou 1).

bitWrite()

Escreve um bit em uma variável.

Sintaxe:

```
bitWrite(x, n, b)
```

Parâmetros

x: variável que deseja ser escrito o bit

n: posição do bit a ser escrito.

b: o valor do bit (0 or 1)

bitSet()

Coloca em 1 o bit de uma variável.

Sintaxe:

```
bitSet(x, n)
```

Parâmetros

x: a variável que se deseja manipular

n: posição do bit que se deseja colocar em 1

bitClear()

Colocar em 0 o bit desejado.

Sintaxe

```
bitClear(x, n)
```

Parametros

x: a variável que se deseja manipular

n: posição do bit que se deseja colocar em 0

bit()

Calcula o valor de um especifico bit.

Sintaxe:

```
bit(n)
```

Parâmetros:

n: a posição do bit que deseja calcular o valor.

Retorno:

O valor do bit.

7.19.5 Funções matemáticas

min(x, y)

Retorna o menor dos dois números.

Parâmetros

x: o primeiro valor a ser comparado

y: o segundo valor a ser comparado

Retorno

O menor dos dois números passados

Exemplo

```
sensVal = min(sensVal, 100); //verifica se sensval é menor que se 100 se maior
                             //garante que não ultrapasse 100
```

max(x, y)

Descrição

Retorna o maior de dois números.

Parâmetros

x: o primeiro valor a ser comparado

y: o segundo valor a ser comparado

Retorno

O maior entre os dois números.

Exemplo

```
sensVal = max(senVal, 20); // garante que sensVal é maior que 20
```

abs(x)

Calcula o valor absoluto de um numero.

Parâmetros

x: o numero

constrain(x, a, b)

Restringe um número para estar dentro de um intervalo.

Parâmetros

x: O numero que deseja restringir;

a: meno valor do intervalo

b: maior valor do intervalo

Retorno

x: se x estiver entre a e b

a: se x é menor que a

b: se x é maior que b

Exemplo

```
sensVal = constrain(sensVal, 10, 150);
                              // limits range of sensor values to between 10 and 150
```

map(value, fromLow, fromHigh, toLow, toHigh)

Re-mapeia um número a partir de um intervalo para o outro. Isto é, um valor de fromLow ficaria mapeada para toLow, um valor de fromHigh ficaria mapeado para toHigh.

Não restringe os valores dentro do intervalo, Por que valores fora do intervalo as vezes são desejados e uteis. A função constrain () pode ser usada antes ou após esta função, se os limites para os intervalos são desejados.

Note que os limites inferiores pode ser maior ou menor do que os "limites superiores" de modo que o map () pode ser usado para inverter uma série de números, por exemplo

```
y = map(x, 1, 50, 50, 1);
```

A função também lida com números negativos, como no exemplo abaixo

```
y = map(x, 1, 50, 50, -100);
```

A função map() usa matemática inteiro por isso não irá gerar frações. Restos fracionários são truncados, e não são arredondados.

Parâmetros

value: numero para mapear

fromLow: limite inferior da faixa atual

fromHigh: limite superior da faixa atual

toLow: limite inferior da faixa futura

toHigh:limite superior da faixa futura

Retorno

O valor Re-mapeado

Exemplo

```
/* Map an analog value to 8 bits (0 to 255) */
void setup() {}
void loop()
{
  int val = analogRead(0);
  val = map(val, 0, 1023, 0, 255);
  analogWrite(9, val);
}
```

pow(base, exponent)

Calcula o valor de um número elevado a uma potência. pow() pode ser usado para elevar um número a uma potência fraccionada. Isto é útil para a geração valores de mapeamento exponencial ou curvas.

Parâmetros

base: o numero da base (float)

exponent: o numero do expoente (float)

Retorno

Resultado da operação (double)

sqrt(x)

Calcula a raiz quadrada de um numero.

Parâmetros

x: o numero, pode ser de qualquer tipo

Retorno

O resultado da operação(double)

sin(rad)

Calcula o seno de um ângulo(radianos). O resultado será entre -1 e 1.

Parâmetros

rad: ângulo em radianos(float)

Retorno

Seno do ângulo.(double)

cos(rad)

Calcula o Cosseno de um ângulo(radianos). O resultado será entre -1 e 1.

Parâmetros

rad: ângulo em radianos(float)

Retorno

Cosseno do ângulo.(double)

tan(rad)

Calcula a tangente de um ângulo (radianos). O resultado estará entre menos infinito e mais infinito.

Parâmetros

rad: ângulo em radianos(float)

Retorno

Tangente do ângulo. (double)

7.19.6 Funções de Conversão

char()

Converte o valor para o tipo char.

Sintaxe

```
char(x)
```

Parâmetros

x: valor de qualquer tipo

Retorno

```
char
```

byte()

Converte um valor para o tipo byte

Sintaxe

```
byte(x)
```

Parâmetros

x: um valor de qualquer tipo

Retorno

```
byte
```

int()

Converte um valor para o tipo int

Sintaxe

```
int(x)
```

Parâmetros

x: um valor de qualquer tipo

Retorno

```
Int
```

word()

Converte um valor em um dado do tipo word ou cria uma word a partir de dois bytes.

Sintaxe

```
word(x)
word(h, l)
```

Parâmetros

x: um valor de qualquer tipo.

h: Parte alta da word

l: parte baixa da word

Retorno

Word

long()

Converte um valor para o tipo de dado long.

Sintaxe

```
long(x)
```

Parâmetros

x: valor a ser convertido

Retorno

```
long
```

float()

Converte um valor para o tipo de dado float.

Sintaxe

```
float(x)
```

Parâmetros

x: valor de qualquer tipo

Retorno

float

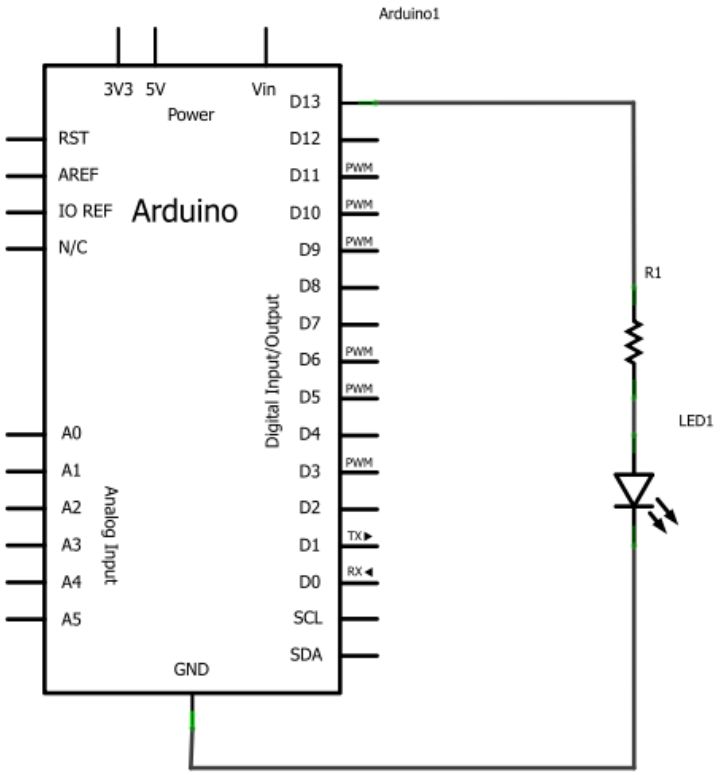
8 PROJETOS

A seguir serão apresentadas algumas aplicações básicas com o ARDUINO, aplicando na pratica as funções estudadas anteriormente. Serão exibidos os conceitos com exemplos assim como a proposta de exercícios.

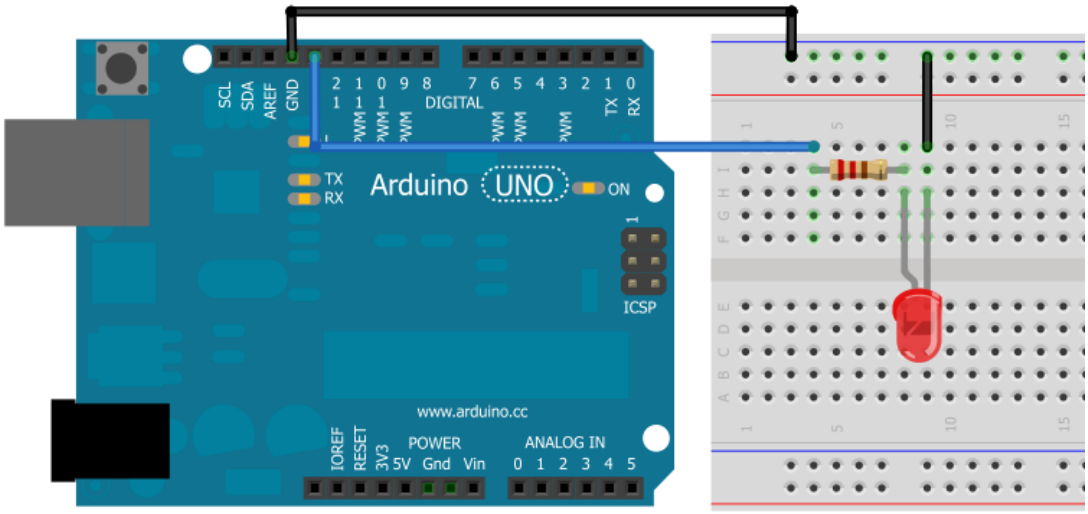
8.1 Saída digital - Piscar LED

O exemplo mais simples para iniciar a programação do Arduino, que pode ser considerado como o conhecido "Hello World" das linguagens de programação, consiste em acionar um LED através de uma saída digital.

A placa Arduino Uno já possui um Led ligado ao pino digital 13 que pode ser utilizado para o teste, a ligação desse LED é exibida a seguir:



Porém podemos, conforme figura a seguir, ligar um LED ao pino 13 no protoboard:



Para configurar o pino como saída digital utiliza a função pinMode(). Inicialmente definiu-se uma variável do tipo **int** com o valor correspondente ao pino que se deseja ligar o LED, no caso o pino 13. Para definir o pino como saída usa-se a constante **OUTPUT**, conforme instrução abaixo:

```
pinMode(led, OUTPUT);
```

Para acionar a saída utiliza-se a função digitalWrite(). Para escrever nível lógico 1, usa-se a constante HIGH e para um nível lógico baixo usa-se a constante LOW, conforme instruções abaixo.

```
digitalWrite(led, HIGH);    // liga led
digitalWrite(led, LOW);     //desliga led
```

A função delay() é utilizada para aguardar 1 segundo entre a mudança de estados. Como a função delay() aguarda por x milissegundos são necessários 1000 ms para se ter 1 segundo:

```
delay(1000);                // aguarda 1 segundo
```

O sketch para a programação do ARDUINO é apresentado em seguida:

```
/*
  Pisca LED
  Liga led por 1 segundo e depois desliga por mais 1 segundo
 */

// Pino 13 tem um led conectado na maioria das placas
int led = 13;      //cria uma variável inteira cchamada led com o numero do pino

// rotina de configuração
void setup() {

  pinMode(led, OUTPUT);     // inicaliza pino do led como saída digital
}

void loop() {
  digitalWrite(led, HIGH);   // liga led
  delay(1000);               // aguarda 1 segundo
  digitalWrite(led, LOW);    //desliga led
  delay(1000);               //aguarda 1 segundo
}
```

Exercício 1:

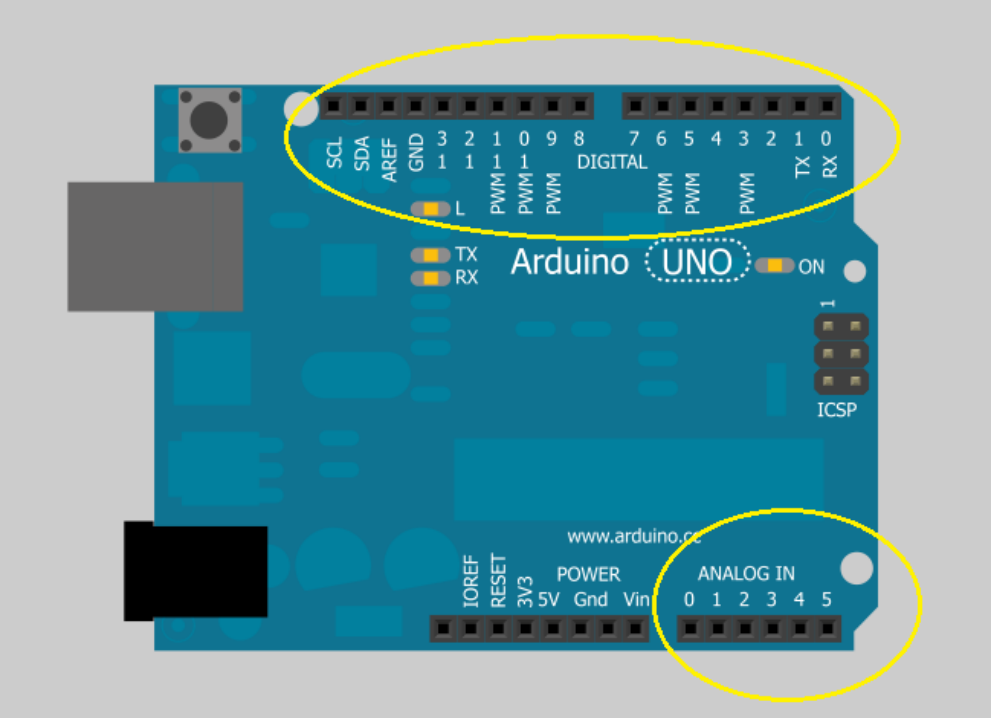
Criar um sketch para acionar 4 LED, conectados aos pinos 10,11,12,13. Os LEDs devem piscar da seguinte forma:

1. Ligar os LEDs 13, 11 e desligar os LEDs 12,10.
2. Aguardar 1 segundo
3. Desligar os LEDs 13, 11 e ligar os LEDs 12,10.
4. Aguardar 1 segundo
5. Repetir o processo.

8.2 Entrada Digital – Lendo tecla

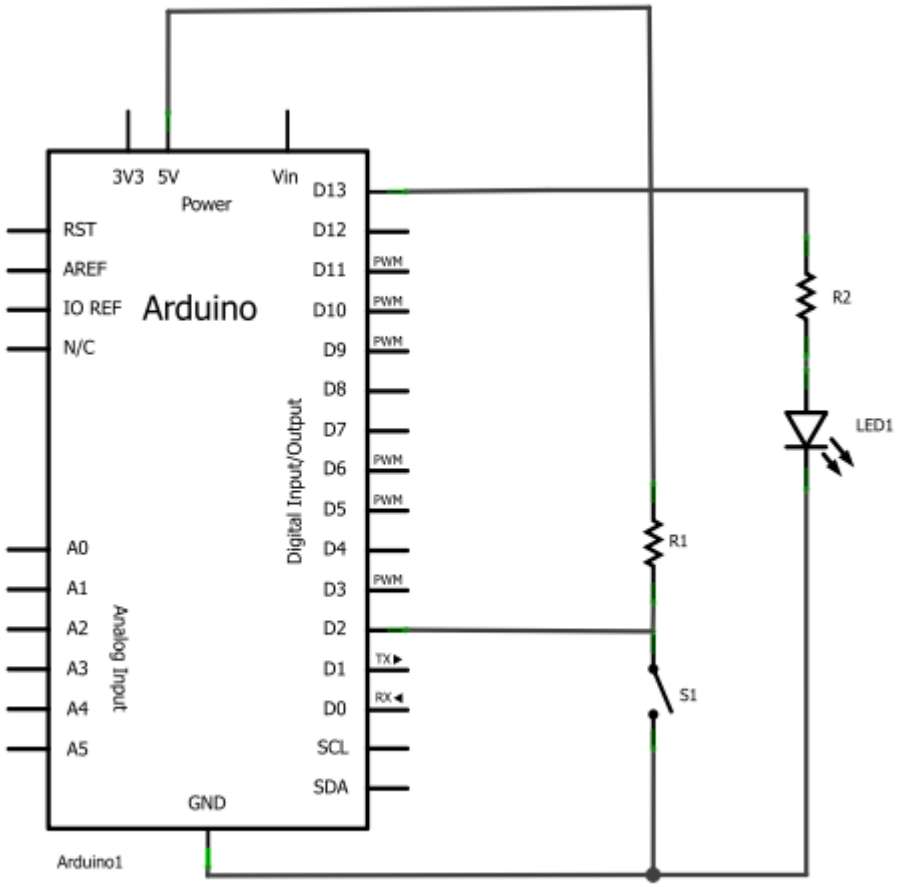
O Arduino pode ler valores de entradas digitais e/ou analógicas. As entradas digitais detectam níveis lógicos nos pinos. As entradas analógicas medem uma faixa de tensão no pino.

Os pinos são divididos em analógicos e digitais, conforme a figura abaixo:



Para ler uma entrada digital utiliza-se a função digitalRead() ela retorna o valor presente no pino. Porém antes de ler o valor presente no pino deve-se configurar o pino correspondente como entrada utilizando a função pinMode().

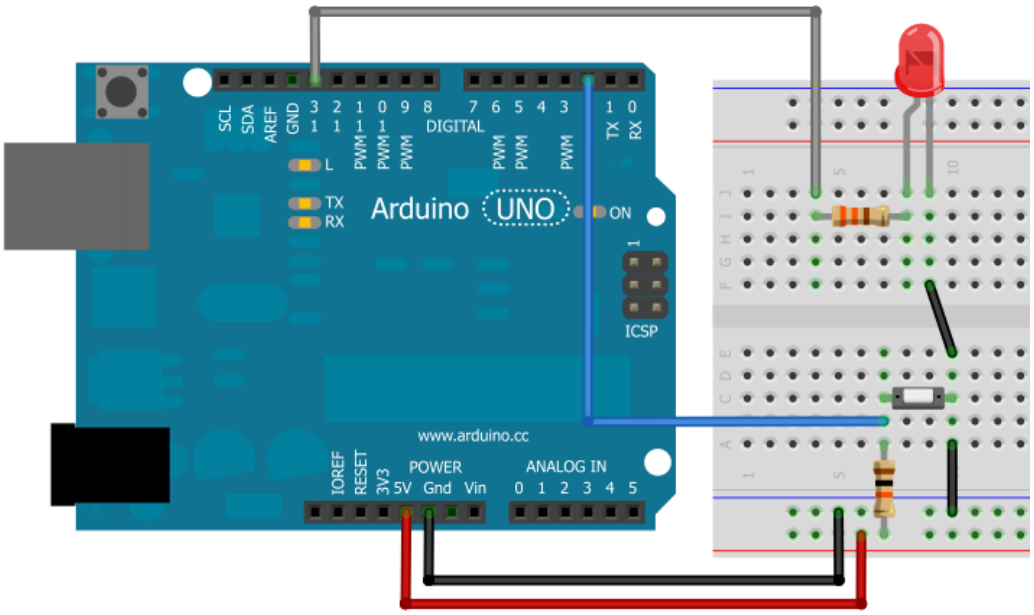
O abaixo é exibido o esquema elétrico para leitura de uma tecla:



A tecla está ligada no pino digital 2, nota-se que existe um resistor ligado a +5V. Este resistor é chamado de resistor de pull – up e tem a função de garantir um nível lógico, no caso 5V, quando a tecla permanecer solta. Para configurar o pino 2 como entrada utiliza-se a seguinte instrução na função setup():

```
pinMode(inputPin, INPUT); // declara pino como entrada
```

O circuito possui um LED ligado ao pino 13 que deve ser configurado como saída digital, conforme visto no capítulo anterior. A montagem do circuito pode ficar conforme a figura abaixo:

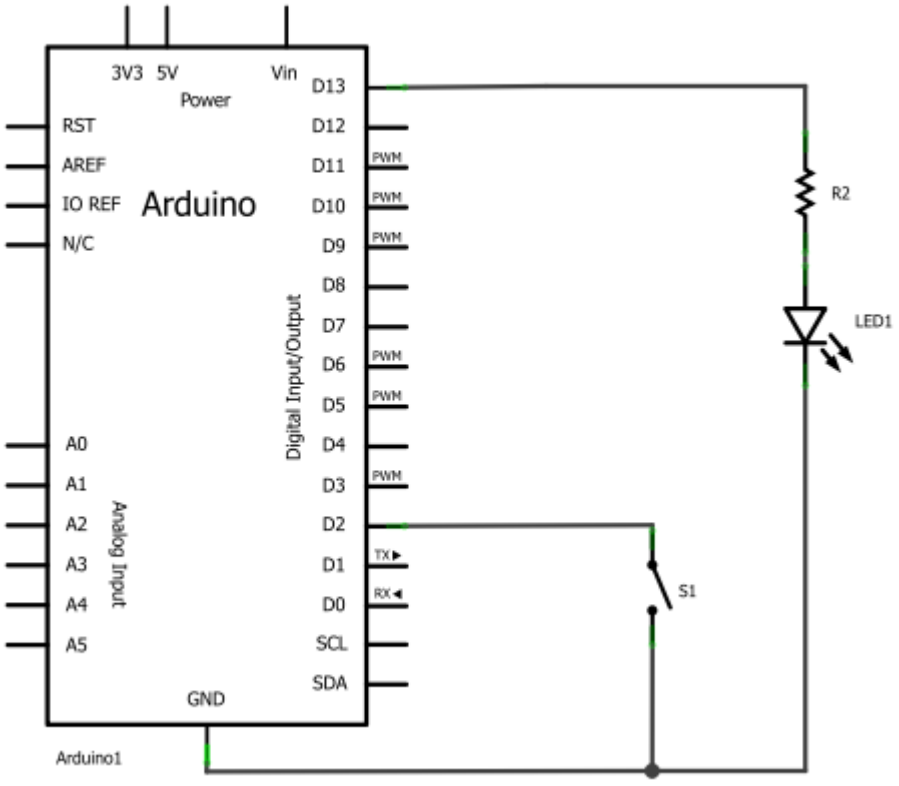


O exemplo consiste em ler a tecla e ligar o LED quando a tecla estiver pressionada, o sketch a seguir mostra a programação para leitura de uma entrada digital:

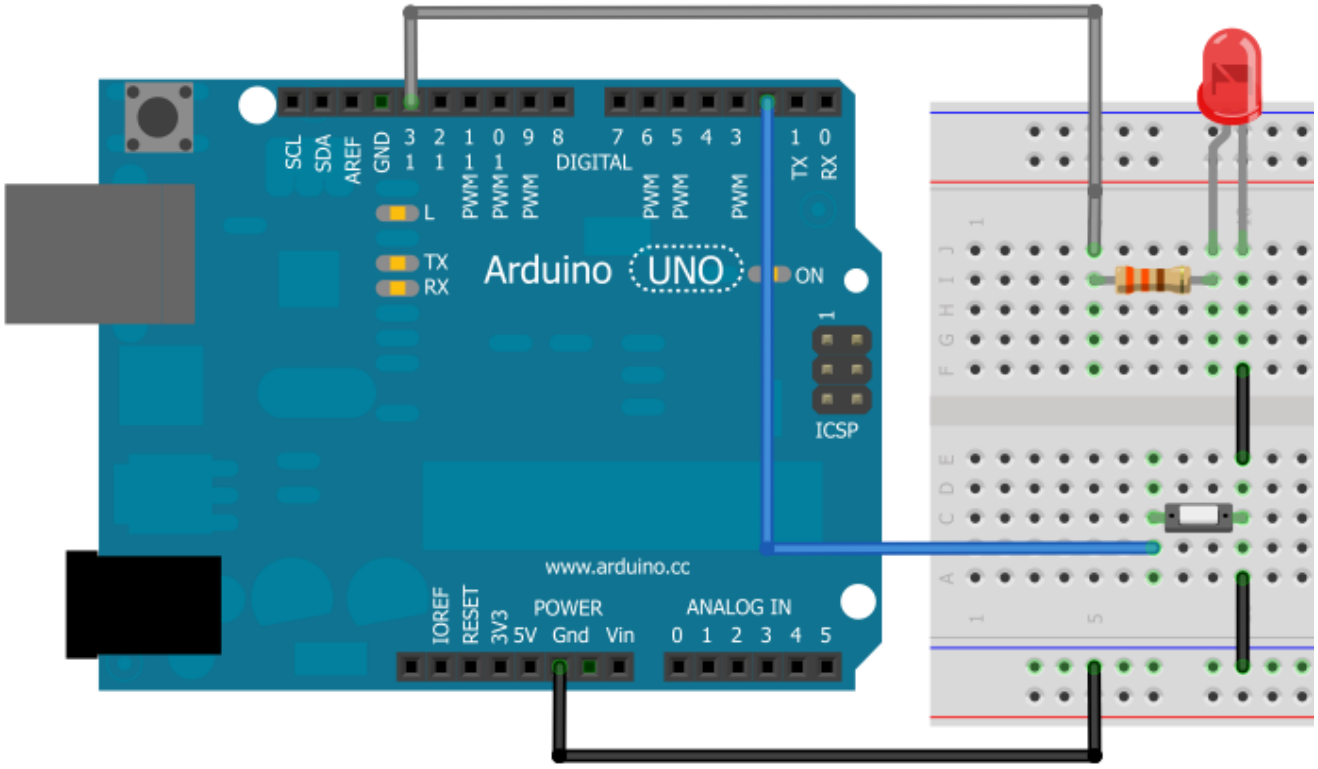
```
/*
Leitura de tecla
O exemplo le uma tecla conectada ao pino 2 e aciona um led conectado ao pino 13
*/

const int ledPin = 13; // cria uma constante com o numero do pino ligado ao LED
const int inputPin = 2; // cria uma constante com o numero do pino conectado a tecla

void setup()
{
      pinMode(ledPin, OUTPUT); // declara o pino do led como saída
      pinMode(inputPin, INPUT); // declara o pino da tecla como entrada
}
void loop()
{
      int val = digitalRead(inputPin); // le o valor na entrada

      if (val == LOW) // se valor está em zero( tecla pressionada)
      {
```

```
            digitalWrite(ledPin, HIGH); // Liga LED indicando tecla pressionada
      }
      else
      {
            digitalWrite(ledPin, LOW); // Desliga led indicando tecla solta
      }
}
```

Os pinos de entrada possuem resistores de "pull-up" internos, dessa forma pode-se ler o estado de uma tecla sem a necessidade de ligar um resistor, como foi feito no exemplo anterior. A figura a seguir exibe o esquema elétrico para ligar uma tecla sem o resistor de pull-up:



O circuito fica mais limpo e mais simples de montar no protoboard, a figura a seguir exibe a ligação no protoboard para teste:



Para configura a entrada com resistor de pull-up ligado, utiliza-se a constante INPUT_PULLUP na função pinMode(), conforme instrução a seguir:

```
pinMode(inputPin, INPUT_PULLUP); //declara o pino da tecla como
                                 //entrada E PULL UP habilitado
```

O sketch completo para o teste da leitura de tecla sem resistor de pull up exibido a seguir:

```
/*
Leitura de tecla
 O exemplo le uma tecla conectada ao pino 2 e aciona um led conectado ao pino 13
*/

const int ledPin = 13; // cria uma constante com o numero do pino ligado ao LED
const int inputPin = 2; // cria uma constante com o numero do pino conectado a tecla
```

```
void setup()
{
  pinMode(ledPin, OUTPUT); // declara o pino do led como saída
  pinMode(inputPin, INPUT_PULLUP); // declara o pino da tecla como entrada
}
void loop()
{
  int val = digitalRead(inputPin); // le o valor na entrada

  if (val == LOW) // se valor está em zero( tecla pressionada)
  {
    digitalWrite(ledPin, HIGH); // Liga LED indicando tecla pressionada
  }
  else
  {
    digitalWrite(ledPin, LOW); // Desliga led indicando tecla solta
  }
}
```

Quando uma tecla é pressionada é gerado oscilações até o nível lógico se estabilizar, essas oscilações são conhecidas como bounce. Esse podem gerar leituras indesejadas fazendo o programa tomar ações que não eram previstas. O processo para eliminar a leitura durante as oscilação é chamado de Debounce.

A técnica de debounce é feita por software e existem muitas maneiras de se resolver este problema, umas delas é apresentada no sketch a seguir:

```
/*
* Debounce sketch
uma tecla é conectada ao pino e um led no pino 13
a lógica de debounce previne leituras indesejadas da tecla
*/

const int inputPin = 2; // pino onde esta a tecla
const int ledPin = 13; // pino onde está conectado o led
const int debounceDelay = 10; // tempo para estabilizar
// rotina de debounce
boolean debounce(int pin)
{
  boolean state;
  boolean previousState;
  previousState = digitalRead(pin); // armazena o estado da tecla
```

```
  for(int counter=0; counter < debounceDelay; counter++)
  {
    delay(1); //aguarda 1 milisegundo
    state = digitalRead(pin); // le a tecla
    if( state != previousState)
    {
      counter = 0; //reinicia contador
      previousState = state; // armazena o valor atual da tecla
    }
  }
  // retorna o estado da tecla
  return state;
}

void setup()
{
  pinMode(inputPin, INPUT);    //configura pino da tecla como entrada
  pinMode(ledPin, OUTPUT);     //configura pino do led como saída
}
void loop()
{
  if (!debounce(inputPin))      //le tecla com rotina de debounce
  {
    digitalWrite(ledPin, HIGH);  //liga led se tecla pressionada
  }
  else
  {
    digitalWrite(ledPin, LOW);  //Desliga led se tecla solta
  }
}
```

A função debounce é chamada passando como parâmetro o numero do pino da tecla que se deseja fazer a leitura. A função retorna o nível lógico presente no pinio após a estabilização.

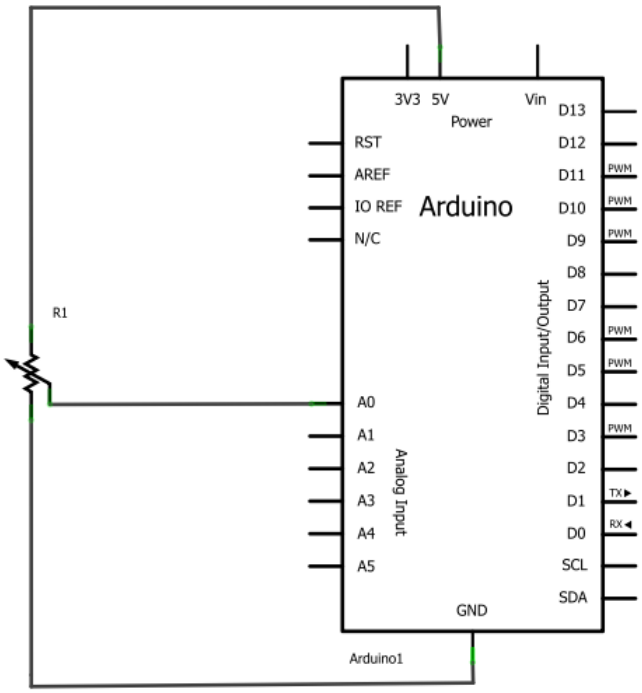
Exercício 2:

Desenvolver um sketch para a leitura de uma tecla e acionar 4 LEDS de saída. A cada vez que a tecla for pressionada deve-se acender um LED e apagar o anterior. Inicialmente os leds encontram-se apagados. Quando tecla pressionada pela primeira vez acende LED1, na segunda vez apaga LED1 e acende LED 2, e assim por diante. Quando o LED4 estiver aceso e a tecla for pressionada apaga o LED4. Quando a tecla for pressionada novamente repetem-se as instruções anteriores.

8.3 Entrada analógica – Lendo o valor em um potenciômetro

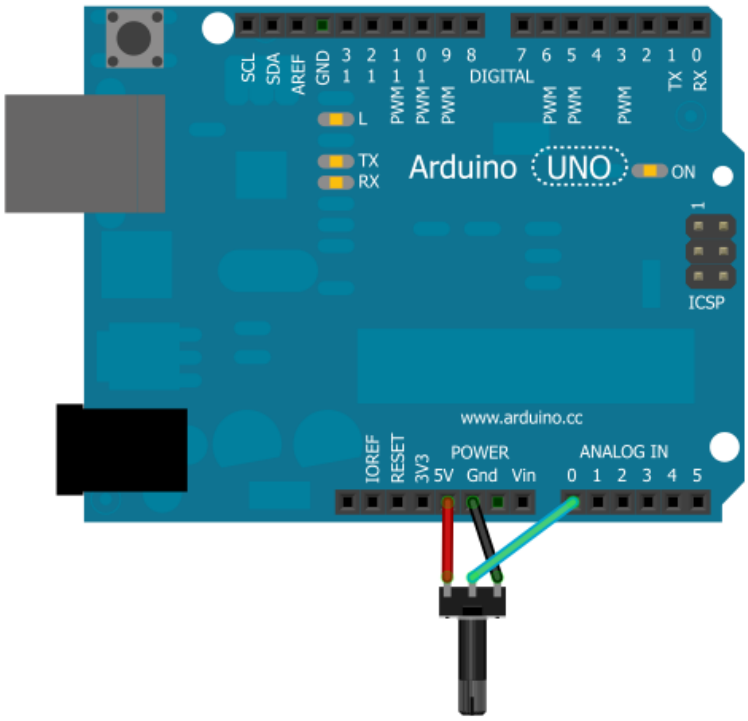
Os microcontroladores Atmega usados para o Arduino contêm embarcados 6 canais de conversão analógica-digital (A/D). O conversor do Arduino tem uma resolução de 10 bits, retornando inteiros entre 0 e 1023. Enquanto a função principal das portas analógicas para a maioria dos usuários do Arduino é ler sensores, as portas analógicas também tem a funcionalidade de entrada/saída de propósito geral (GPIO) (da mesma forma que as portas digitais 0~13). Consequentemente, se o usuário precisar mais entradas/saídas de propósito geral, e todas as portas analógicas não estiverem em uso, estas podem ser usadas para GPIO.

Para iniciar o estudo da entrada analógica, a maneira mais simples e rápida é ligando um potenciômetro a uma entrada analógica, conforme esquema apresentado em seguida:



Se girarmos o cursor do potenciômetro, alteramos a resistência em cada lado do contato elétrico que vai conectado ao terminal central do botão. Isso provoca a mudança na proximidade do terminal central aos 5 volts ou GND o que implica numa mudança no valor analógico de entrada. Quando o cursor for levado até o final da escala, teremos, por exemplo, zero volt a ser fornecido ao pino de entrada do Arduino e, assim, ao lê-lo obtém-se 0. Quando giramos o cursor até o outro extremo da escala, haverá 5 volts a ser fornecido ao pino do Arduino e, ao lê-lo, teremos 1023. Em qualquer posição intermediária do cursor, teremos um valor entre 0 e 1023, que será proporcional à tensão elétrica sendo aplicada ao pino do Arduino.

A ligação no Arduino UNO pode ser feita conforme a figura abaixo:



O exemplo a seguir lê o valor no potenciômetro. O tempo que o LED permanece ligado ou desligado depende do valor obtido pelo analogRead().

```
/*
 * Entrada analógica
 * Liga e desliga um LED conectado ao pino digital 13. O tempo
 * que o LED permanece ligado ou desligado depende do valor
 * obtido pelo analogRead(). No caso mais simples, conecta-se
 * um potenciômetro ao pino analógico 0.
 */

int potPin = 0;     // selecione o pino de entrada ao potenciômetro
int ledPin = 13;    // selecione o pino ao LED
int val = 0;        // variável a guardar o valor proveniente do sensor
```

```
void setup() {
  pinMode(ledPin, OUTPUT);  // declarar o pino ledPin como saída
}

void loop() {
  val = analogRead(potPin);    // ler o valor do potenciômetro
  digitalWrite(ledPin, HIGH);  // ligar o ledPin
  delay(val);                  // pausar o programa por algum tempo
  digitalWrite(ledPin, LOW);   // desligar o ledPin
  delay(val);                  // pausar o programa por algum tempo
}
```

Exercício 3:

Elaborar um sketch para ler um potenciômetro e acender 8 leds nas saídas digitais que representarão o nível de tensão na entrada analógica como um bargraph. Quando entrada for zero volts todos os leds estarão apagados e quando for 5 Volts todos estarão acesos e qualquer valor dentro desse intervalo deve acender os leds proporcionalmente.

8.4 Comunicação Serial

A comunicação serial provê um caminho fácil e flexível para a placa ARDUINO interagir com um computador ou outro dispositivo. Outro ponto interessante é que a IDE do Arduino, possui um Monitor serial que exibe os dados enviados pelo Arduino ou envia dados para o Arduino. Esse monitor pode ser utilizado para exibir mensagens durante o teste do projeto.

Uma comunicação serial envolve hardware e software, o hardware faz a ligação elétrica entre o Arduino e o dispositivo que ele está comunicando. O software envia ordenadamente os dados. A biblioteca Arduino auxilia na complexidade da comunicação serial tornando fácil e simples o seu uso.

Através da comunicação serial pode-se criar supervisórios no computador para ler sinais do Arduino e exibir na tela de um computador, pode-se enviar comandos através de uma interface gráfica no computador para o Arduino executar determinadas tarefas.

Para exibir textos e números enviados do Arduino, no computador, deve-se inicialmente configura a comunicação serial através da função Serial.begin() na função setup(), e usar a função Serial.print() para enviar o texto ou valores para o computador.

A função Serial.begin() seleciona o baudrate da comunicação serial. O baudrate determina quantos bits por segundo é enviando na transmissão de um dado. Para a comunicação com um computador usa-se os seguintes valores: 300, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600 ou 115200.

A sintaxe é exibida a seguir:

Serial.begin(speed);

Exemplo:

Serial.begin(9600);

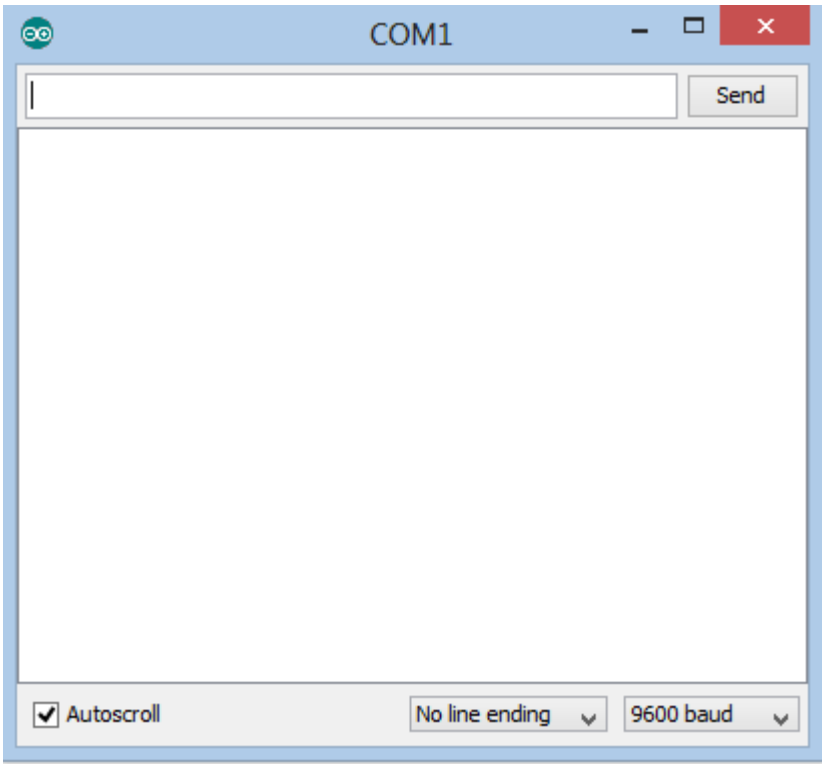
A função print envia textos pela serial. O texto deve ser colocado dentro de aspas duplas, por exemplo:

Serial.print(“olá, mundo”);

Para enviar números, o valor exibido dependerá do tipo da variável. Por exemplo, para enviar uma variável do tipo int, chamada numero, usa-se a seguinte função:

Serial.print(numero);

Para visualizar os valores no computador pode-se utilizar o terminal serial integrado a IDE conforme figura a seguir:

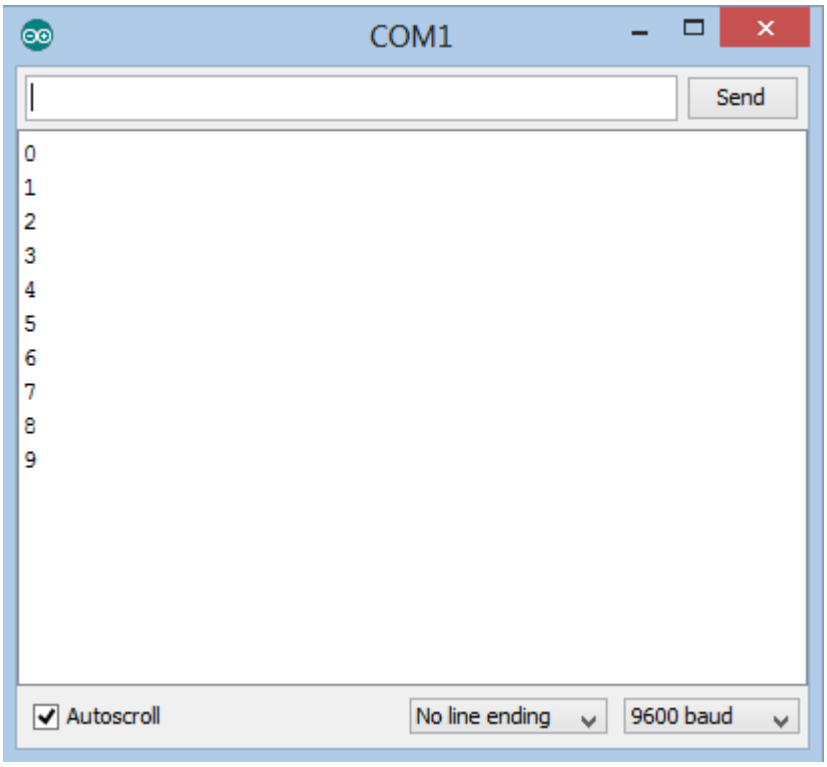


Exemplo A – Enviando números para o computador

O sketch a seguir exibe como a comunicação serial pode ser utilizada para enviar dados para um computador.

```
/*
* comunicação Serial
* Envia números pela comunicação serial
* para ser exibido no computador
*/
void setup()
{
  Serial.begin(9600); // seleciona velocidade para 9600 bps
}
int number = 0;           //variável para ser enviada


void loop()
{
  Serial.println(number); // envia numero em noiva linha
  delay(500);             // aguarda 0,5 segundo
  number++;               // incrementa o valor do numero.
}
```

Serão exibidos no terminal serial os números no terminal conforme a figura a seguir:



Exemplo B – Enviando números em diferentes formatos

Para imprimir um texto simplesmente usa a função: Serial.print("texto a ser enviado!"). para enviar uma nova linha após o envio do texto, usa-se a função: Serial.println("texto a ser enviado");

Para imprimir valores numéricos pode ser um pouco mais complicado. A forma que um valor do tipo byte ou inteiro são exibidos depende do tipo de variável e dos parâmetros de formatação. A linguagem ARDUINO auxilia nessa tarefa possibilitando a exibir um valor em diferente tipo de dados.

Por exemplo, enviando um char, um byte e um int de um mesmo valor a saída não será a mesma.

Segue um exemplo especifico todas as variáveis a seguir tem valores similares:

```
char asciiValue = 'A'; // ASCII A has a value of 65
char chrValue = 65; // an 8 bit signed character, this also is ASCII 'A'
byte byteValue = 65; // an 8 bit unsigned character, this also is ASCII 'A'
int intValue = 65; // a 16 bit signed integer set to a value of 65
float floatValue = 65.0; // float with a value of 65
```

Os seguintes parâmetros podem ser utilizados para enviar valores em diferentes tipos de dados:

Serial.print(valor, BIN) - envia no formato binário

Serial.print(valor, OCT) -envia no formato octal

Serial.print(valor, DEC) – envia no formato decimal

Serial.print(78, HEX) – envia no formato hexadecimal

Serial.println(1.23456, 0) - envia numero flutuante sem casa decimal

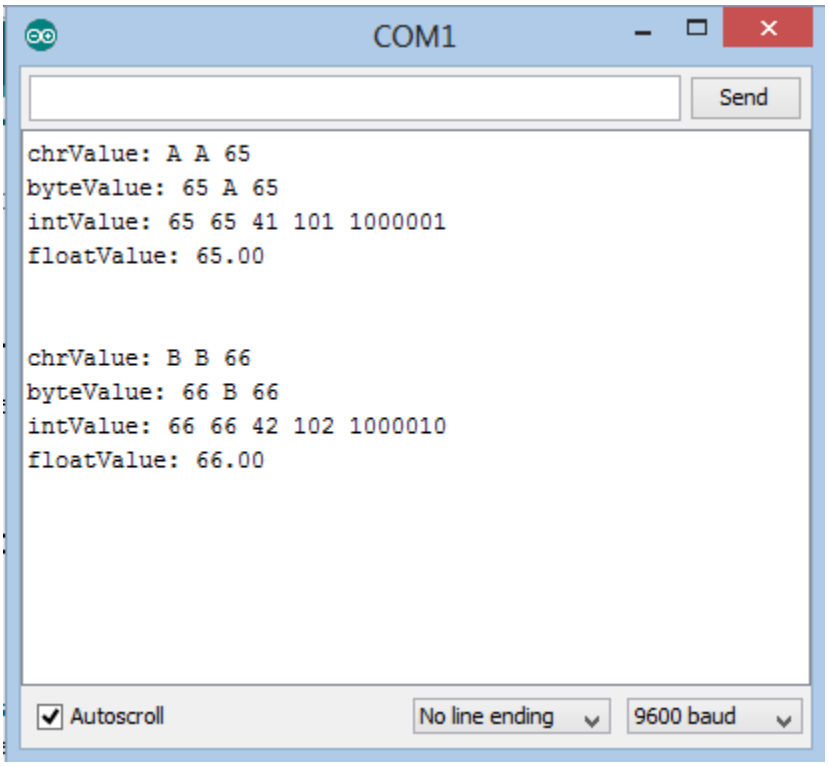
Serial.println(1.23456, 2) - envia numero flutuante com duas casa decimais

Serial.println(1.23456, 4) - envia numero flutuante com 4 casas decimais

O sketch a seguir envia dados em diferentes formatos pela comunicação serial:

```
/*
* Comunicação serial - formatar dados
 * Envia valores
 */
char chrValue = 65; //valores para ser enviados
byte byteValue = 65;
int intValue = 65;
float floatValue = 65.0;

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


void loop()
{
  Serial.print("chrValue: ");
  Serial.print(chrValue);
  Serial.print(" ");
  Serial.write(chrValue);
  Serial.print(" ");
  Serial.print(chrValue,DEC);
  Serial.println();

  Serial.print("byteValue: ");
  Serial.print(byteValue);
  Serial.print(" ");
  Serial.write(byteValue);
  Serial.print(" ");
  Serial.print(byteValue,DEC);
  Serial.println();

  Serial.print("intValue: ");
  Serial.print(intValue);
```

```
  Serial.print(" ");
  Serial.print(intValue,DEC);
  Serial.print(" ");
  Serial.print(intValue,HEX);
  Serial.print(" ");
  Serial.print(intValue,OCT);
  Serial.print(" ");
  Serial.print(intValue,BIN);
  Serial.println();

  Serial.print("floatValue: ");
  Serial.print(floatValue);
  Serial.println();
  Serial.println();
  Serial.println();

  delay(5000); // aguarda 5 segundo
  chrValue++; // incremnta para o próximo valor
  byteValue++;
  intValue++;
  floatValue +=1;
}
```



Exemplo C – Enviando números para o Arduino

São enviados valores em ASCII para o Arduino, esses valores podem ser convertidos em valores numéricos de de uma maneira bem simples, bastando conhecer a tabela ASCII.

O valores de 0 a 9 possuem o valor 48 a 57 na tabela ASCII. Desta forma para converter um valor ASCII de 0 a 9 em um valor numérico basta subtrair o valor 48, ou seja subtrair o caractere ‘0’ em ASCII.

O sketch a seguir exibe uma maneira simples de receber valores em ASCII e converter em valores numéricos:

```
/*
* Comunicação Serial - Enviar dados para o Arduino
 * PISCA o led com o valor proporcional ao recebino pela serial
 */
const int ledPin = 13; //pino do led
int blinkRate=0; // valor para piscar

void setup()
{
  Serial.begin(9600); // Inicializa serial com 9600 bps
  pinMode(ledPin, OUTPUT); //configura pino do led como saida
}

void loop()
{
  if ( Serial.available()) // verifica se há caracter disponivel na serial
  {
    char ch = Serial.read();  //le o caracter
    if( isDigit(ch) ) // verifica se valor está entre 0 e 9
    {
      blinkRate = (ch - '0'); // converte caracter em um valor numérico
      blinkRate = blinkRate * 100; // atualiza valor da piscagem
    }
  }
  blink();//chama rotina para piscar led
}

void blink()
{
  digitalWrite(ledPin,HIGH);
  delay(blinkRate);
  digitalWrite(ledPin,LOW);
  delay(blinkRate);
}
```

Exercício 4:

Desenvolver um sketch para ligar os leds conforme o número enviado pela serial. Se enviado o numero ‘1’ deve-se ligar o led1 e assim por diante. Quando enviado o mesmo numero na segunda vez deve-se desligar o led correspondente.

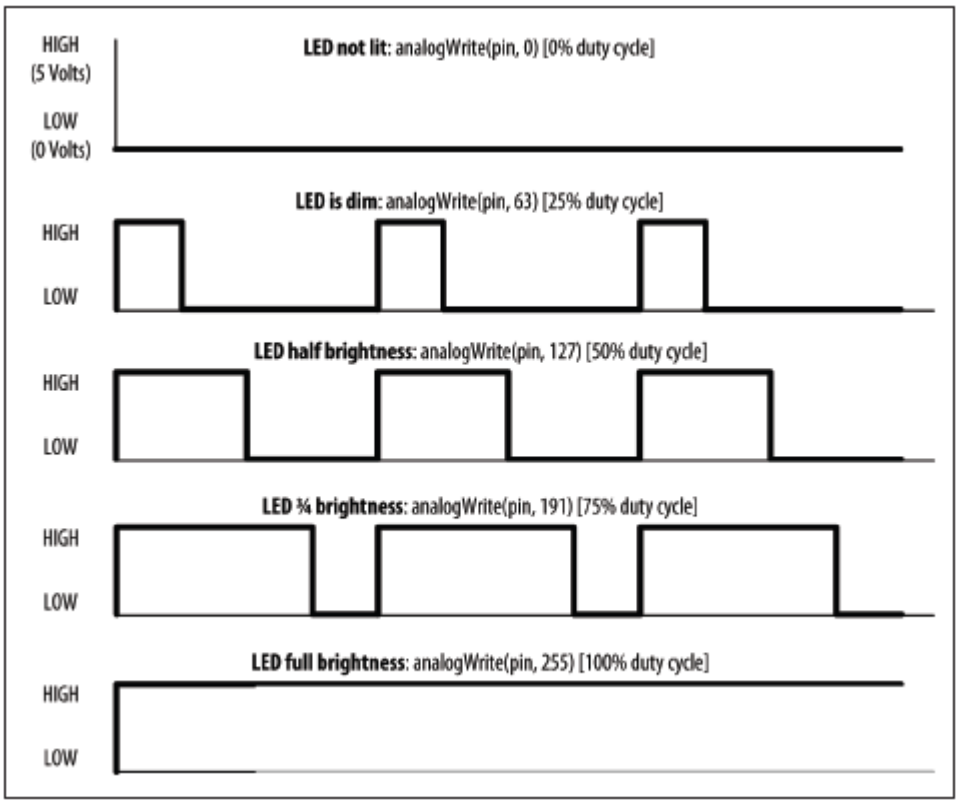
8.5 6.5. Saída PWM

A saída PWM no Arduino pode ser feita através da função analogWrite(), que pode ser utilizada para controle diversos, como por exemplo a intensidade do brilho de um led conectado ao ARDUINO.

A técnica de PWM (do inglês Pulse width modulation) consiste em emular um sinal analógico através de pulsos digitais.

PWM consiste na variação proporcional dos pulsos, mantendo a frequência constante. Saídas de baixo nível de sinal analógico são conseguidas com pulso curtos de nível lógico alto, já saídas com maior nível de sinal analógico são conseguidos através de períodos maiores de pulso em nível alto.

A figura a seguir exibe a modulação PWM para variar a intensidade de um led:



O Arduino possui pinos específicos com essa função. Na placa ARDUINO UNO, pode se utilizar os pinos 3,5,6,9,10 e 11 para esta função.

A função analogWrite deve ser utilizada da seguinte forma:

Sintaxe:

analogWrite(pino, valor);

Onde o parâmetro pino corresponde ao pino que será gerado o sinal PWM e valor corresponde ao duty cycle, ou seja, o valor que permanecerá em nível alto o sinal.

O valor deve ser de 0 a 255 onde 0 a saída permanece sempre em nível zero e 225 a saída permanece sempre em nível alto.

O sketch a seguir exibe como controlar a intensidade de um led através da leitura de um sinal analógico em um potenciômetro.

```
/*
  PWM
 controla a luminosidade de um led conforme o valor presente
 em um potenciometro
 */

int ledPin = 13;      // pino do led
int analogPin = 3;   // pino para leitura do potenciometro
int val = 0;         // variable to store the read value

void setup()
{
  pinMode(ledPin, OUTPUT);   // configura pino como saída
}

void loop()
{
  val = analogRead(analogPin);   // le o valor analógico
  analogWrite(ledPin, val / 4);  // aciona led com o valor anlogico lido divido por 4
}
```

Exercício 5

Desenvolver um sketch para controlar o brilho do led através de duas teclas onde uma aumenta o brilho do led e outra diminui o brilho do led.

9 REFERÊNCIAS

1. http://arduino.cc/en/Reference/HomePage