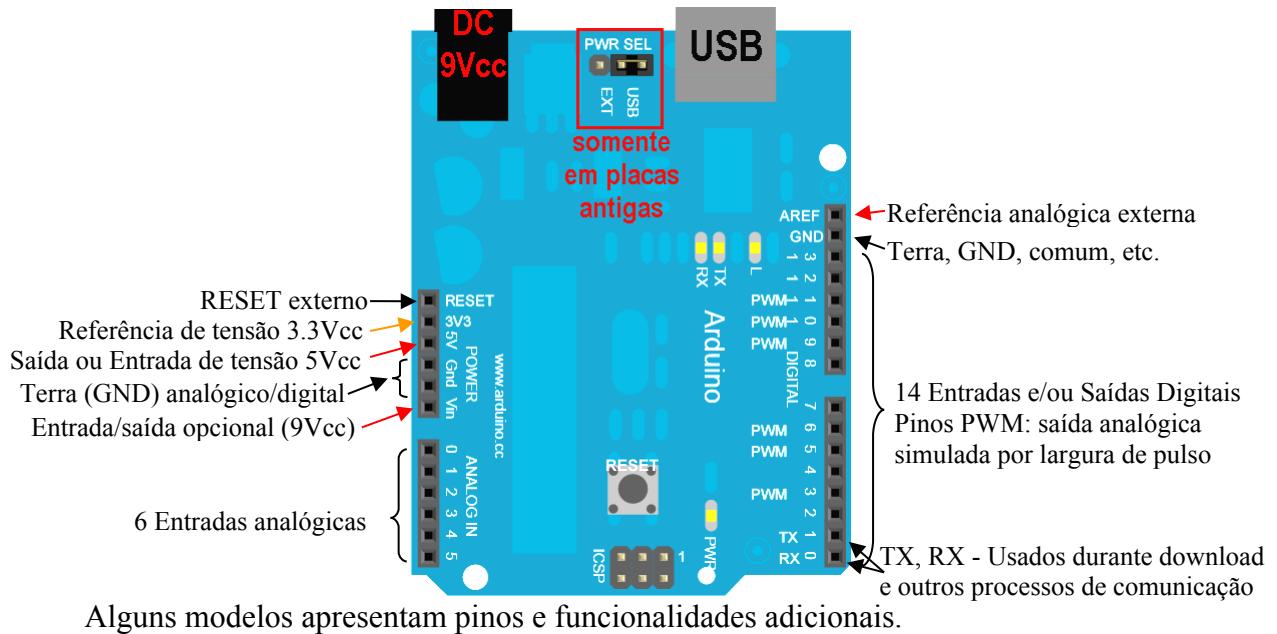


Referência rápida do Arduino

Versão revisada em Janeiro/2024



Leitura de diagramas esquemáticos

Em eletrônica, diagramas esquemáticos permitem que se descreva um circuito de uma forma que possa ser compreendida por engenheiros e outros técnicos. As ilustrações mais detalhadas para descrever a montagem de circuitos eletrônicos utilizam diagramas elétricos esquemáticos, mas em alguns casos não é prático desenhar um diagrama esquemático quando se quer simplesmente documentar um experimento, e nesses casos os diagramas de blocos podem ser até mais úteis.

Na engenharia é comum o uso de simbologia específica como uma forma rápida de se capturar a essência de um projeto, para documentá-lo e reconstruí-lo posteriormente, ou mesmo passá-lo a outra pessoa. Para gerar tais documentações são estabelecidas normas para representação dos diversos componentes individuais onde são criados símbolos que representam abstrações de seu próprio formato, ou de sua essência. Por exemplo, o capacitor é formado por duas placas de metal separadas por um isolante (ar, vidro, plástico, etc.). Estabeleceu-se então o seu símbolo como:

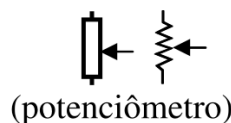
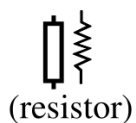
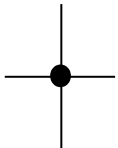


(capacitor)

Outro exemplo semelhante é o indutor, formado por um fio de cobre espiralado em formato cilíndrico, cujo símbolo pode ser visto à esquerda.



Conexões entre componentes são geralmente feitas utilizando-se fios ou as próprias conexões da placa de circuito impresso. Quando dois fios estão conectados, utiliza-se um ponto para representar o local em que as duas linhas se cruzam, como na ilustração à esquerda. Os símbolos a seguir compõem uma lista de componentes comuns com seu significado.



Pode-se encontrar variações desses símbolos (por exemplo, os dois tipos de símbolos para resistores). Caso necessário, pode-se ainda consultar en.wikipedia.org/wiki/Electronic_symbol para uma lista mais completa de símbolos eletrônicos. Por convenção, diagramas são desenhados da esquerda para a direita. O circuito de um rádio, por exemplo, seria desenhado com sua antena à

esquerda, seguido pelo caminho que o sinal de rádio percorre até o alto-falante, desenhado à direita.

Introdução à Plataforma Arduino

O Arduino é uma plataforma de computação física de fonte aberta, com base em uma placa de circuito simples, com conexões para entrada e saída (input/output, ou I/O). Além disso, existe um ambiente de desenvolvimento implementado pela linguagem Processing (www.processing.org). O Arduino pode ser utilizado para desenvolver objetos interativos independentes, ou conectado ao computador através de ambientes de programação (como Flash, Processing, VVVV, ou Max/MSP). O Integrated Development Environment (IDE), um ambiente de desenvolvimento código aberto para o Arduino, pode ser baixado gratuitamente de www.arduino.cc.

O Arduino é diferente de outras plataformas presentes no mercado em razão dos seguintes fatores:

- ✓ É um ambiente multiplataforma; pode ser executado no Windows, Macintosh e Linux.
- ✓ Tem por base o IDE de programação Processing, ambiente de desenvolvimento fácil de ser utilizado e que costuma ser empregado por artistas e designers.
- ✓ Pode ser programado utilizando-se um cabo USB, sem necessidade de uma porta serial. Esse recurso é útil, uma vez que muitos computadores modernos não têm portas seriais.
- ✓ Apresenta Hardware e software de fonte aberta - pode-se fazer o download do diagrama de circuito, comprar todos os componentes e criar um original Arduino, não sendo necessário pagar qualquer valor (\$\$\$) aos criadores originais.
- ✓ O Hardware é barato. Os modelos USB já custam abaixo dos €20 e em modelos mais antigos é muito fácil substituir um processador queimado, além de não custar mais do que €5 ou US\$4. Justamente por isso, eventuais erros fatais não acarretarão grandes prejuízos.
- ✓ Há uma comunidade ativa de usuários, com muitas pessoas que podem ajudar.
- ✓ O Arduino Project foi desenvolvido em um ambiente educacional; portanto, é ideal para iniciantes que desejam resultados rápidos.

Este material foi selecionado a partir de documentação original dos criadores do Arduino, e pretende auxiliar iniciantes a compreender os benefícios que podem ser obtidos ao aprender a utilizar a plataforma Arduino e ao adotar sua filosofia. O modo como o texto procura explicar tópicos não é o mesmo utilizado em textos para engenharia.

O Arduino foi criado inicialmente para ensinar Design de Interação, uma disciplina de design que coloca a prototipagem no centro de sua metodologia. Há muitas definições para Design de Interação, e a mais adequada para o propósito desse texto é:

Design de Interação é o projeto de qualquer experiência interativa.

No mundo atual, o Design de Interação preocupa-se com a criação de experiências significativas entre seres humanos e objetos. É uma forma de se explorar a criação de experiências válidas entre os seres humanos e a tecnologia. O Design de Interação encoraja projetos que utilizem um processo iterativo, com base em protótipos de fidelidade crescente. Essa abordagem pode ser estendida para incluir a prototipagem aplicada à tecnologia; em especial, a prototipagem de eletrônicos. O campo específico do Design de Interação envolvido no Arduino é a Computação Física (ou Design de Interação Física).

Computação Física

A Computação Física utiliza elementos de eletrônica na prototipagem de novos materiais para designers e artistas. Ela envolve o projeto de objetos interativos que podem se comunicar com seres humanos utilizando sensores e atuadores controlados por um comportamento implementado simulado por software, executado dentro de um microcontrolador (um pequeno computador ou chip individual).

No passado, o uso de elementos de eletrônica significava lidar com engenheiros o tempo todo e criar circuitos desenvolvendo um componente de cada vez. Nem todas as pessoas com capacidade criativa eram capazes de se envolver diretamente com esse meio pois a maioria das ferramentas adequadas a esse tipo de desenvolvimento era destinada a engenheiros e exigia vasto

conhecimento técnico. Nos anos recentes, microcontroladores tornaram-se mais baratos e de mais fácil utilização, permitindo a criação de ferramentas melhores.

O progresso alcançado com o Arduino significa que essas ferramentas ficaram mais próximas e simples para o iniciante, permitindo que essas pessoas construam seus projetos depois de apenas dois ou três dias de trabalho. Designers e artistas podem rapidamente aprender o básico de eletrônica e uso de sensores com o Arduino, criando seus protótipos com pouco investimento.

Linguagem de Programação do Arduino

As instruções básicas aceitas pela linguagem do Arduino estão listadas a seguir. Para uma referência mais detalhada, consulte:

arduino.cc/en/Reference/HomePage.

Estrutura

Todos os aplicativos para o Arduino são denominados *sketches*. Um *sketch* no Arduino é composto de duas partes:

void setup() - Local em que deve ser colocado o código de inicialização - instruções que preparam a placa antes do início do loop principal do *sketch*.

void loop() - Seção que contém o código principal de um *sketch* e que deve apresentar um conjunto coerente de instruções a serem repetidas seguidas vezes até que a placa seja desligada.

Ex:

```

/*-----
LED piscante, 1.0 Hz no pino 2
-----*/
void setup()                // ações únicas, sem repetição
{
    pinMode(2,OUTPUT);      // define pino 2 como saída
}
void loop()                 // loop contínuo, sem interrupção
{
    digitalWrite(2,HIGH);   // pino 2 em "1" (LED ligado)
    delay(500);             // espera 500 ms
    digitalWrite(2,LOW);   // pin 2 em "0" (LED desligado)
    delay(500);             // espera 500 ms
}

```

Símbolos especiais

A linguagem do Arduino inclui alguns símbolos que devem ser empregados em linhas de código, comentários e blocos de código:

; (ponto e vírgula)

Toda linha de código completa é encerrada por um ponto e vírgula. Essa sintaxe permite formatar o código livremente. Pode-se colocar mais de uma instrução por linha, desde que separadas por um ponto e vírgula. (*Não indicado, pois poderá dificultar o entendimento do código.*)

Exemplo: **delay(100);**

{ } (chaves)

Essa notação é utilizada para marcar blocos de código. Por exemplo, quando se escreve um código para a função **loop()**, deve-se utilizar chaves antes e depois do código.

Exemplo:

```

void loop() {
    Serial.println("ciao");
}

```

Comentários

São as partes do código que serão descartadas pelo Arduino na execução, mas extremamente

úteis para recordar a funcionalidade do trecho do código, ou para explica-lo a outras pessoas. Há duas formas de criação de comentários no Arduino:

```
// de linha única: o texto até o fim da linha será ignorado
/* de várias linhas:
você pode escrever
um poema inteiro aqui
*/
```

Constantes

A linguagem do Arduino inclui várias palavras-chave predefinidas com valores especiais.

HIGH e **LOW** utilizadas, por exemplo, quando você deseja ligar ou desligar um pino.

INPUT e **OUTPUT** utilizadas para definir um pino específico como entrada ou saída.

TRUE e **FALSE** indicam se uma condição ou expressão é verdadeira ou falsa, respectivamente.

Variáveis

Variáveis são áreas separadas na memória do Arduino para armazenar dados que serão utilizados e manipulados em seu *sketch*. Como o nome sugere, variáveis podem ser modificadas quantas vezes forem necessárias. Como o Arduino contém um processador muito simples, quando uma variável é declarada deve-se também especificar seu tipo. Com isso, “informa-se” ao processador o tamanho do valor que será armazenado na variável.

Tipos de dados disponíveis

boolean (ou **bool**) - Tipo booleana. Pode ter os valores verdadeiro (**TRUE**) ou falso (**FALSE**).

char - Armazena um único caractere, como A. Da mesma forma que qualquer computador, o Arduino armazena esse valor como um número, ainda que você veja um texto. Quando caracteres são utilizados para armazenar números, eles podem armazenar valores de **-128** a **127**.

NOTA: Há dois grandes grupos de caracteres disponíveis a sistemas de computadores: ASCII e Unicode. ASCII é um conjunto de 127 caracteres, utilizado, entre outras coisas, para transmitir textos entre terminais seriais e sistemas de computadores trabalhando em modo compartilhado, como mainframes e minicomputadores. Unicode, por sua vez, é um conjunto muito maior de valores utilizados por sistemas operacionais mais atuais, para representar caracteres em muitos idiomas diferentes. ASCII ainda é útil, por exemplo, para transferência de pequenas quantidades de informações, em idiomas como italiano e inglês, que utilizam caracteres latinos, numerais arábicos e símbolos datilográficos comuns para pontuação.

byte - Armazena um valor entre **0** e **255**. Assim como no caso de **char**, variáveis do tipo **byte** utilizam apenas um byte de memória.

int - Utiliza dois bytes da memória para armazenar um número, na faixa de **-32.768** e **32.767**. Um dos tipos de dados mais comumente utilizados no Arduino.

unsigned int - Utiliza 2 bytes como em **int**, mas o prefixo **unsigned** significa que não pode armazenar números negativos. Seu alcance fica na faixa entre **0** a **65.535**.

long - O dobro de um **int**. Armazena números de **-2.147.483.648** a **2.147.483.647** e ocupa 4 bytes de memória.

unsigned long - Versão não sinalizada de **long**. Utiliza 4 bytes e armazena números entre **0** e **4.294.967.295**.

float - Tipo de dado capaz de armazenar valores equivalentes a números com ponto flutuante (parte decimal). Um **float** utiliza 4 bytes da RAM. As funções que podem utilizá-lo também consomem muita memória, por isso *o uso de variáveis float é recomendado somente quando estritamente necessário*.

double - Número ponto flutuante de precisão dupla. Utiliza até 8 bytes da RAM para cada

variável. Valor máximo: $1.7976931348623157 \times 10^{308}$!

string - Conjunto de caracteres ASCII (array ASCII) utilizados para armazenar informações textuais (pode-se utilizar uma string para enviar uma mensagem por meio de uma porta serial, ou para apresentá-la em um monitor LCD). Quanto ao armazenamento, uma string utiliza um byte para cada caractere, mais um caractere nulo utilizado pelo Arduino para indicar que o fim da linha de texto foi atingido. As duas linhas seguintes são equivalentes:

```
char string1[] = "Arduino"; // 7 caracteres + 1 caractere nulo
char string2[8] = "Arduino"; // igual à linha anterior
```

array - Lista de variáveis que podem ser acessadas por meio de um índice. Um array é utilizado para criar tabelas de valores que podem ser acessados com facilidade. Por exemplo, para controlar o brilho de um LED com 5 diferentes níveis, pode-se criar cinco variáveis (**light01**, **light02**, etc.) e armazenar um valor em cada, mas utilizando-se um array fica mais simples:

```
int light[5] = {0, 20, 50, 75, 100};
```

A criação de variáveis no formato array precisa somente dos símbolos [] e {}.

Estruturas de controle

O Arduino inclui palavras-chave para controlar o fluxo lógico dos *sketchs*.

If ... else - Essa estrutura é responsável pela tomada de decisões em um programa. **if** deve ser seguido por uma questão, especificada como uma expressão entre parênteses. Se a expressão resultar verdadeira, o que vier depois dela será executado. Se falsa, o bloco de código que segue **else** será executado. É possível utilizar apenas **if**, sem especificar uma cláusula **else**.

Exemplo:

```
if (val == 1) {
    digitalWrite(LED, HIGH);
}
```

for - Permite que você repita um bloco de código por um número especificado de vezes.

Exemplo:

```
for (int i = 0; i < 10; i++) {
    Serial.print("OI!");
}
```

switch ... case - Enquanto a instrução **if** funciona como uma bifurcação que oferece duas opções no programa, **switch case** se parece mais com uma enorme rotatória. Ela permite que o programa receba várias orientações, dependendo do valor de uma variável, sendo ótima para manter um código organizado, uma vez que substitui listas (ou sequências) de instruções **if**.

Exemplo:

```
switch (sensorValue) {
    case 23:
        digitalWrite(13, HIGH); // somente se sensorValue == 23
        break;
    case 46:
        digitalWrite(12, HIGH); // somente se sensorValue == 46
        break;
    default:
        // sensorValue com outros valores
        digitalWrite(12, LOW);
        digitalWrite(13, LOW);
}
```

while - Semelhante a **if**, **while** executará um bloco de código enquanto determinada condição for verdadeira.

Exemplo:

```
// pisca o LED enquanto o valor do sensor estiver abaixo de 512
sensorValue = analogRead(1);
while (sensorValue < 512) {
    digitalWrite(13, HIGH);
    delay(100);
    digitalWrite(13, LOW);
    delay(100);
    sensorValue = analogRead(1);
}
```

do... while - Idêntica à **while**. A única diferença é que, aqui, o código será executado antes de a condição ser avaliada. Essa estrutura é utilizada para que um bloco de código seja executado ao menos uma vez antes de conferir a condição.

Exemplo:

```
do {
    digitalWrite(13, HIGH);
    delay(100);
    digitalWrite(13, LOW);
    delay(100);
    sensorValue = analogRead(1);
}
while (sensorValue < 512);
```

break - Esse termo permite que o programa saia de um loop e continue a execução do código que aparece depois dele. **break** também é utilizado para separar seções distintas de uma instrução **switch case**.

Exemplo:

```
// pisca o LED enquanto o valor do sensor estiver abaixo de 512
do {
    // sai do loop se um botão é pressionado
    if (digitalRead(7) == HIGH)
        break;
    digitalWrite(13, HIGH);
    delay(100);
    digitalWrite(13, LOW);
    delay(100);
    sensorValue = analogRead(1);
} while (sensorValue < 512);
```

continue - Quando utilizado dentro de um loop, **continue** permite que o programa pule o resto do código e faça com que a condição seja testada novamente.

Exemplo:

```
for (light = 0; light < 255; light++)
{
    // pula as intensidades entre 140 e 200
    if ((x > 140) && (x < 200))
        continue;
    analogWrite(PWMPin, light);
    delay(10);
}
```

return - Interrompe a execução de uma função, retornando-a. Pode-se também utilizar esse comando para retornar um valor de dentro da função. Por exemplo, uma função de nome **computeTemperature()** pode retornar seu resultado para a seção do código que a invocou com a sequência de código a seguir:

```
int computeTemperature() {
```

```
int temperature = 0;
temperature = (analogRead(0) + 45) / 100;
return temperature;
}
```

Aritmética e fórmulas

O Arduino pode ser utilizado para realizar cálculos complexos utilizando uma sintaxe especial. + e - funcionam como você aprendeu na escola. A multiplicação é representada com por * e a divisão, por /. Há também um operador adicional, chamado "módulo" (%), que retorna o resto da divisão de um inteiro.

Pode-se utilizar quantos níveis de parênteses forem necessários para agrupar as expressões. Ao contrário do que você aprendeu na escola, colchetes e chaves estão reservados a outros propósitos (índices e blocos de arrays, respectivamente).

Exemplos:

```
a = 2 + 2;
light = ((12 * sensorValue) - 5) / 2;
remainder = 3 % 2; // retorna 1
```

Operadores de comparação

Quando se especifica condições ou testes para instruções **if**, **while** e **for**, estes são os operadores que podem ser utilizados:

```
== igual a
!= não igual a (diferente de)
< menor do que
> maior do que
<= menor ou igual a
>= maior ou igual a
```

Operadores booleanos

Operadores booleanos serão utilizados quando se quer combinar várias condições. Por exemplo, para verificar se o valor vindo de um sensor está entre 5 e 10, deve-se escrever:

```
if ((sensor == 5) && (sensor <=10))
```

A linguagem do Arduino oferece três operadores booleanos:

```
&&; representa e lógico (AND).
||; representa ou lógico (OR).
!; representa não lógico (NOT).
```

Operadores compostos

Operadores compostos são operadores especiais utilizados para tornar o código mais conciso quando temos de realizar operações comuns, como incrementar o valor de uma variável.

Por exemplo, para incrementar a variável **value** em uma unidade, pode-se escrever:

```
value = value + 1;
```

Utilizando um operador composto, essa linha se tornará:

```
value++;
```

Incremento e decremento (-- e ++)

Incrementam e decrementam determinado valor em uma unidade. Observe que (**cuidado!**): **i++**, incrementa o valor de **i** em uma unidade e avalia um resultado equivalente a **i+1**, e **++i**, por sua vez, avalia o valor de **i** e só então faz o incremento. O mesmo se aplica a **i--** e **--i**.

+=, -=, *= e /= Esses operadores facilitam o uso em algumas expressões. As duas expressões

a seguir são equivalentes:

```
a = a + 5;
a += 5;
```

Funções de entrada e saída

O Arduino inclui funções para a manipulação de entradas e saídas.

pinMode(#pino, modo) - Reconfigura um pino digital para ser utilizado como entrada ou saída. (**modo** -> **OUTPUT**, **INPUT** ou **INPUT_PULLUP**)

Exemplo:

```
pinMode(7, INPUT); // define o pino 7 como entrada
```

OBS: O modo **INPUT_PULLUP** ativa a impedância interna de pullup do microcontrolador para o pino configurado (entre 40KΩ e 41KΩ no ATMEGA328).

digitalWrite(#pino, valor) – Determina a condição de um pino digital. **valor** pode ser **HIGH** ou **LOW** (1 ou 0 podem ser utilizados respectivamente no lugar de **HIGH** ou **LOW**). **digitalWrite** só tem efeito se **#pino** for definido como **OUTPUT** (saída) com **pinMode**.

Exemplo:

```
pinMode(8, OUTPUT);
digitalWrite(8, HIGH); // leva a "1" (HIGH) o pino digital 8
```

digitalRead(#pino) - Lê o estado de um pino de entrada, retornando **HIGH** (1) se o pino recebe voltagem, ou **LOW** (0) se não há voltagem aplicada.

Exemplo:

```
val = digitalRead(7); // lê o valor do pino 7, atribuindo-o a val
```

analogWrite(#pino, valor) - Estabelece a taxa PWM em um dos pinos marcados PWM. **#pino** pode ser 11, 10, 9, 6, 5 ou 3. **value** pode ser um número entre 0 e 255, representando a escala entre 0 e 5V da tensão de saída.

Exemplo:

```
analogWrite(9, 128); // Define luminosidade do LED no pino 9 em 128
```

analogRead(#pino) - Lê a tensão aplicada a um pino de entrada analógica (**A0** a **A5** no Arduino UNO) e retorna um número entre 0 e 1023, representando voltagens entre 0 e 5V

Exemplo:

```
val = analogRead(A0); // lê o valor da entrada analógica A0,
// atribuindo a val (val deve ser long)
```

IMPORTANTE

O valor retornado por **analogRead()** é de 10 bits (0 a 1023) e se for utilizada uma variável de 8 bits (declarada como **int**) para armazenar esse valor, ele será truncado para 8 bits antes de ser armazenado (os dois bits mais significativos serão perdidos), causando erros. Existem algumas formas para evitar a perda desses valores mantendo-se a variável de 8 bits na leitura.

Exemplo 1: *dividir o valor lido por 4, reduzindo-se a faixa de 0 a 1023 para a faixa de 0 a 255*

```
val = analogRead(A0) / 4;
```

Exemplo 2: *mapear o valor lido na faixa de 0 a 1023 para a faixa de 0 a 255 com a função **map()***

```
val = map(analogRead(A0), 0, 1024, 0, 255);
```

shiftOut(dataPin, clockPin, bitOrder, valor) - Envia dados para um registrador de deslocamento, dispositivo utilizado para expandir o número de saídas digitais. Esse protocolo utiliza um pino para dados e um para o clock. **bitOrder** indica a ordem dos bytes (do menos significativo para o mais significativo) e valor é o byte em si que será enviado.

Exemplo: (**dataPin**, **clockPin** e **LSBFIRST** - variável da função - são pre-definidos)

```
shiftOut(dataPin, clockPin, LSBFIRST, 255);
```


unsigned long pulseIn(#pino, valor) - Mede a duração de um pulso vindo de uma das entradas digitais. Isso é útil quando se quer, por exemplo, ler sensores infravermelhos ou acelerômetros que emitem seus valores como pulsos de duração diferente.

Exemplo:

```
time = pulseIn(7, HIGH); // mede o tempo que o próximo
                        // pulso permanece high
```

Funções de tempo

O Arduino inclui funções capazes de medir o tempo transcorrido e também pausar o *sketch*.

unsigned long millis() - Retorna o número de milissegundos transcorridos desde que o *sketch* teve início.

Exemplo:

```
duration = millis() - lastTime; // computa tempo transcorrido
                                // desde "lastTime"
```

delay(ms) - Pausa o programa pelo tempo especificado em milissegundos.

Exemplo:

```
delay(500); // interrompe o programa por meio segundo
```

delayMicroseconds(us) - Pausa o programa pelo tempo especificado em microssegundos.

Exemplo:

```
delayMicroseconds(1000); // espera 1 milissegundo
                          // equivale a delay(1);
```

Funções matemáticas

O Arduino inclui muitas funções matemáticas e trigonométricas comuns:

min(x, y) - Retorna o menor valor entre *x* e *y*.

Exemplo: `val = min(10, 20);` // val será 10

max(x, y) - Retorna o maior valor entre *x* e *y*.

Exemplo: `val = max(10, 20);` // val será 20

abs(x) - Retorna o módulo de *x*. Se *x* == 5 ou se *x* == -5, **abs(a)** retorna sempre 5.

Exemplo: `val = abs(-5);` // val será 5

constrain(x, a, b) - Retorna o valor de *x* restrito ao intervalo entre *a* e *b*. Se *x* for menor que *a*, ela retorna *a*. Se *x* for maior que *b*, ela retorna *b*.

Exemplo:

```
val = constrain(analogRead(0), 0, 255); // rejeita valores lidos
                                        // maiores do que 255
```

map(valor, fromLow, fromHigh, toLow, toHigh) - Mapeia um valor no intervalo entre *fromLow* e *fromHigh*, e interpola no intervalo entre *toLow* e *toHigh*. Esse recurso é muito útil para processar valores de sensores analógicos.

Exemplo:

```
val = map(analogRead(0), 0, 1023, 100, 200); // mapeia o valor do pino
                                              // analógico [0] a um
                                              // valor entre 100 e 200
```

double pow(base, expoente) - Retorna o resultado da potenciação de um número (*base*) por um valor inteiro específico (*expoente*).

Exemplos:

```
double x = pow(y, 32); // define x como y elevado à 32ª potência
```

double sqrt(x) - Retorna a raiz quadrada de um número. Exemplo:

```
double a = sqrt(1138); // aproximadamente 33,73425674438
```

double sin(rad) - Retorna o seno de um ângulo especificado em radianos. Exemplo:

```
double sine = sin(2); // aproximadamente 0,99929737891
double cos(rad) - Retorna o cosseno de um ângulo especificado em radianos. Exemplo:
double cosine = cos(2); // aproximadamente -0,41614685058
double tan(rad) - Retorna a tangente de um ângulo especificado em radianos.
double tangent = tan(2); // aproximadamente -2,18503975868
```

Funções de números aleatórios

Caso haja necessidade de gerar números aleatórios, pode-se utilizar o gerador de números pseudoaleatórios do Arduino.

randomSeed(semente) - Essa função redefine o gerador de números pseudoaleatórios do Arduino. Ainda que a distribuição dos números retornados por **random()** seja essencialmente aleatória, a sequência que ela oferece é previsível. Por isso, deve-se redefinir o gerador utilizando um valor aleatório. Caso haja algum pino analógico desconectado, ele deve captar ruídos variados do ambiente (como ondas de rádio, raios cósmicos, interferência eletromagnética de celulares e luzes fluorescentes e assim por diante) que podem ser utilizados nesse caso.

Exemplo:

```
randomSeed(analogRead(5)); // randomiza utilizando valor do pino 5
```

long random(max) e **long random(min, max)** - Retorna um valor de inteiro long pseudoaleatório, entre min e max -1. Se o valor mínimo não for especificado, seu limite será 0.

Exemplos:

```
long randnum = random(0, 100); // um número entre 0 e 99
long randnum = random(11); // um número entre 0 e 10
```

Comunicação serial

Como visto no capítulo 5, o Arduino pode se comunicar com outros dispositivos por meio da porta USB utilizando um protocolo de comunicação serial. A seguir, as funções seriais.

Serial.begin(taxa) - Prepara o Arduino para iniciar a transferência de dados seriais. No caso do monitor serial do IDE do Arduino, geralmente será utilizada uma taxa de 9.600 bits por segundos (bps), mas há outras taxas de transferência disponíveis, normalmente inferiores a 115.200 bps.

Exemplo: **Serial.begin(9600);** // estabelece 9600bps

Serial.print(dados) e **Serial.print(dados, codificação)** - Envia dados para a porta serial. A codificação é opcional, mas se não for especificada os dados serão tratados como texto simples.

Exemplos:

```
Serial.print(75); // imprime "75"
Serial.print(75, DEC); // igual ao comando anterior
Serial.print(75, HEX); // "4B" (75 em notação hexadecimal)
Serial.print(75, OCT); // "113" (75 em notação octal)
Serial.print(75, BIN); // "1801611" (75 em notação binária)
Serial.print(75, BYTE); // "K" (o byte que representa
// 75 no conjunto ASCII)
```

Serial.println(dados) e **Serial.println(dados, codificação)** - Funciona como **Serial.print()**, exceto adiciona um retorno de carro e um avanço de linha (r\n), como se você tivesse digitado os dados e pressionado Return ou Enter.

Exemplos:

```
Serial.println(75); // imprime "75\r\n"
Serial.println(75, DEC); // igual ao comando anterior
Serial.println(75, HEX); // "4B\r\n"
Serial.println(15, OCT); // "113\r\n"
```

```
Serial.println(75, BIN); // "1091011\r\n"  
Serial.println(75, BYTE); // "K\r\n"
```

int Serial.available() - Retorna o número de dados não lidos, disponíveis na porta serial para leitura por meio da função **Serial.read()**. Depois que você tiver lido todos os dados disponíveis, **Serial.available()** retornará 0 (zero) até que novos dados cheguem à porta serial.

Exemplo:

```
int count = Serial.available();
```

int Serial.read() - Busca um byte dos dados seriais entrantes.

Exemplo:

```
int data = Serial.read();
```

Serial.flush(); - Como os dados podem chegar à porta serial mais rápido do que seu programa é capaz de processá-los, o Arduino armazenará todos os dados entrantes em um buffer. Caso você tenha de limpar o buffer para preenchê-lo com novos dados, utilize a função flushQ.

Exemplo: **Serial.flush();**

REFERÊNCIA

Esse documento foi escrito tendo-se como base o livro *Getting Started with Arduino*, de Massimo Banzi - Editora O'Reilly - 2011.