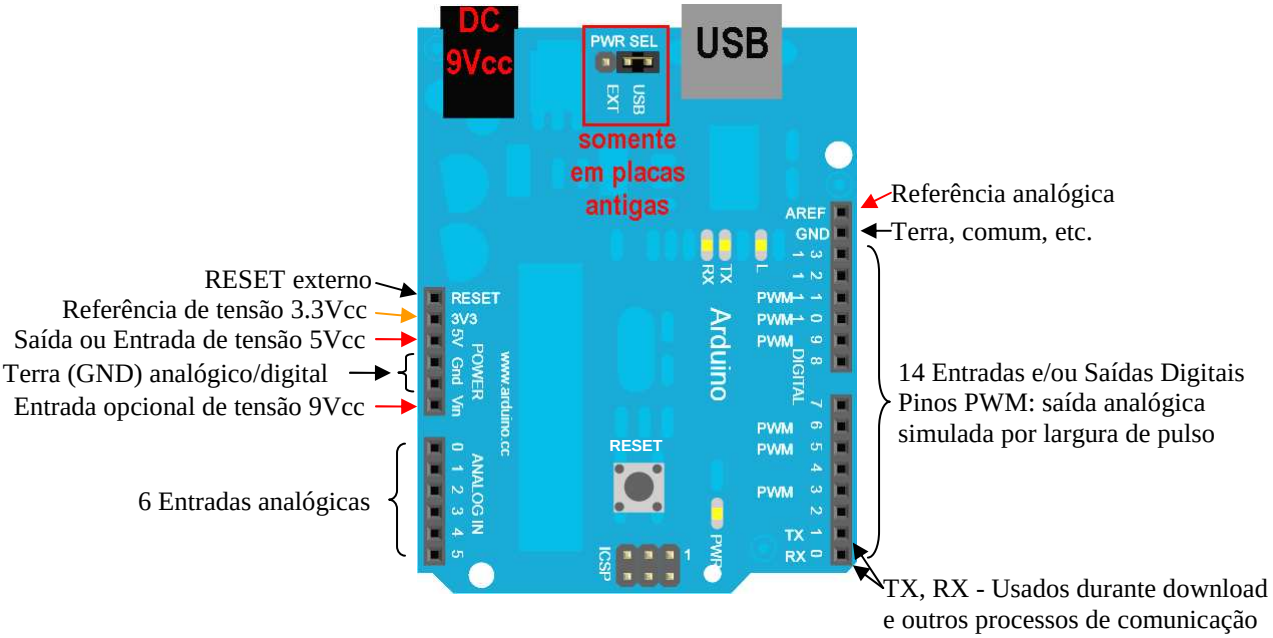
Referência rápida do Arduino



Leitura de diagramas esquemáticos

As ilustrações mais detalhadas para descrever a montagem de circuitos, utilizam diagramas elétricos esquemáticos. Mas em alguns casos não é prático desenhar um diagrama esquemático quando se quer simplesmente documentar um experimento.

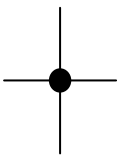
Problemas semelhantes surgem, mais cedo ou mais tarde, em todas as áreas de atuação. Na engenharia é comum o uso de simbologia específica como uma forma rápida de se capturar a essência de um projeto, para documentá-lo e reconstruí-lo posteriormente, ou mesmo passá-lo a outra pessoa.

Em eletrônica, diagramas esquemáticos permitem que se descreva um circuito de uma forma que possa ser compreendida por engenheiros e outros técnicos. Componentes individuais são representados por 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 ar ou plástico; assim, seu símbolo é:

(capacitor)

(indutor)

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

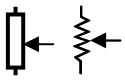


Conexões entre componentes são geralmente feitas utilizando-se fios ou as próprias conexões da placa de circuito impresso. Quando dois fios estão conectados, utiliza-se um ponto para representar o local em que as duas linhas se cruzam, como na ilustração à esquerda. Os símbolos a seguir compõem uma lista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

(resistor)

(diodo)

(LED)



(potenciômetro)

(inteerruptor)

(LDR)

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.ee.

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

- ✓ Trata-se de um ambiente multiplataforma; ele 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.
- ✓ É um Hardware e software de fonte aberta - se você quiser, pode fazer o download do diagrama de circuito, comprar todos os componentes e criar seu próprio Arduino, sem ter de pagar 'nada aos criadores originais.
- ✓ O Hardware é barato. A placa USB custa cerca de €20 (atualmente, algo em torno de U$35) e substituir um chip queimado é muito fácil, além de não custar mais do que €5 ou US$4. Justamente por isso, seus eventuais erros não acarretarão grandes prejuízos.
- ✓ Há uma comunidade ativa de usuários, com muitas pessoas que podem ajudá-lo.
- ✓ 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 fáceis de serem utilizados, 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. Com o Arduino, um designer ou artista pode rapidamente aprender o básico de eletrônica e sensores e começar a criar 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* do Arduino é composto de duas partes:

`void setup()` - Local em que você coloca seu 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 seu *sketch* e que deve apresentar um conjunto 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 sem repetição
      {
            pinMode(2,OUTPUT); // define pino 2 como saída
      }
void loop() // loop initerrupto
      {
            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 instrução (linha de código) é encerrada por um ponto e vírgula. Essa sintaxe permite que você formate seu código livremente. Você pode até colocar duas instruções na mesma linha, desde que as separe por um ponto e vírgula. (Esteja atento, pois isso dificultará a leitura de seu código.)

Exemplo: **`delay(100);`**

`{ }` (chaves)

Essa notação é utilizada para marcar blocos de código. Por exemplo, quando você escreve código para a função **`loop()`**, tem de utilizar chaves antes e depois do código.

Exemplo:

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

Comentários

Porções do texto ignoradas pelo processador do Arduino, mas extremamente úteis para que você se recorde do que fez nesse trecho do código, ou para explicar essa funcionalidade a outras pessoas.

Há dois tipos 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 um conjunto de 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 nomeadas da memória do Arduino nas quais se podem armazenar dados que serão utilizados e manipulados em seu *sketch*. Como o nome sugere, variáveis podem ser modificadas quantas vezes você quiser. Como o Arduino contém um processador muito simples, quando você declara uma variável, deve também especificar seu tipo, ou seja: dizer ao processador o tamanho do valor que você deseja armazenar.

Tipos de dados disponíveis:

`boolean` - Pode ter um de dois valores: verdadeiro ou falso.

`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 declare variáveis **`float`** apenas quando estritamente necessário.

`double` - Número ponto flutuante de precisão dupla. Valor máximo: $1.7976931348623157 \times 10^{308}$!

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

```
char stringl[] = "Arduino"; // 7 caracteres + l 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, caso você queira armazenar níveis diferentes de brilho a serem utilizados por um LED, pode criar seis variáveis, nomeadas **`light01`**, **`light02`** e assim por diante. Se utilizar um array fica mais simples, como:

```
int light[6] = {0, 20, 50, 75, 100};
```

A palavra "array" não é utilizada na declaração da variável: os símbolos **`[]`** e **`{}`** são definidores.

Estruturas de controle

O Arduino inclui palavras-chave para controlar o fluxo lógico de seu *sketch*.

`if...else` - Essa estrutura é responsável pela tomada de decisões em seu programa. **`if`** deve ser seguido por uma questão, especificada como uma expressão entre parênteses. Se a expressão for 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 ao seu programa, switch case se parece mais com uma enorme rotatória. Ela permite que seu programa receba várias orientações, dependendo do valor de uma variável, sendo ótima para manter seu código organizado, uma vez que substitui listas de instruções **if** .

Exemplo:

```
switch (sensorValue) {
     case 23:
          digitalwrite{13, HIGH);
          break;
     case 46:
          digitalwrite(12, HIGH);
          break;
     default: //se nenhum case é executado, executa-se esta linha
          digitalWrite(12, LOW);
          digitalwrite(13, LOW);
}
```

while - Semelhantemente 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(l);
while (sensorValue < 512) {
     digitalwrite(13, HIGH);
     delay(100);
     digitalwrite(13, HIGH);
     delay(10e);
     sensorValue = analogRead(l);
}
```

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 quando você deseja que seu bloco de código seja executado ao menos uma vez antes de conferir a condição.

Exemplo:

```
do {
     digitalwrite(13, HIGH);
     delay(100);
     digitalWrite(13, HIGH);
```

```
     delay(100);
     sensorValue = analogRead(l);
}
while (sensorValue < 512);
```

break - Esse termo permite que você 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(l);
} while (sensorValue < 512);
```

continue - Quando utilizado dentro de um loop, continue permite que você 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. Você também pode utilizar esse comando para retornar um valor de dentro da função.

Por exemplo, se você tiver uma função de nome `computeTemperature()` e quiser retornar seu resultado para a seção do código que a invocou, poderia escrever algo como:

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

Aritmética e fórmulas

Você pode utilizar o Arduino 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 um operador adicional, chamado "módulo" (**%**), que retorna o resto da divisão de um inteiro. Você pode 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 l
```

Operadores de comparação

Quando você 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, caso se queira verificar se o valor vindo do sensor está entre 5 e 10, deve-se escrever:

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

Há três operadores:

e (AND), representado por **`&&`**;
ou (OR), representado por **`||`**;
não (NOT), representado por **`!`**.

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++`, você incrementa o valor de `i` em uma unidade e avalia um resultado equivalente a `i+1`;
`++i`, por sua vez, avalia o valor de `i` e só então faz o incremento. O mesmo se aplica a `--`.

`+=,-=,*=` e **`/=`**

Esses operadores facilitam o uso de certas 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. Você já viu algumas delas nos programas deste livro.

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

Exemplo:

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

`digitalWrite(pino,valor)` - Liga ou desliga um pino digital. **`digitalWrite`** só tem efeito se **`pino`** for definido como saída através de **`pinMode`**. (**`valor`** -> **`LOW`** ou **`HIGH`**)

Exemplo:

```
digitalWrite(8, HIGH); // leva a “1” (HIGH) o pino digital 8
```

`intdigitalRead(pino)` - Lê o estado de um pino de entrada, retornando `HIGH` se o pino recebe voltagem, ou `LOW` se não há voltagem aplicada.

Exemplo:

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

`intanalogRead(pino)` - Lê a tensão aplicada a um pino de entrada analógica e retorna um número entre 0 e 1023, representando voltagens entre 0 e 5V

Exemplo:

```
val = analogRead(B); // lê o valor da entrada analógica 8,
                     // atribuindo a val
```

`analogWrite(pino,valor)` - Altera a taxa PWM em um dos pinos marcados PWM. `pin` 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 a luminosidade do LED em pin9 em 128
```

`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:

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

`unsigned long pulseln(pino, valor)` - Mede a duração de um pulso vindo de uma das entradas digitais. Isso é útil quando queremos, 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 l milissegundo
```

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); // agora, val é 10
```

`max(x,y)` - Retorna o maior valor entre `x` e `y`. Exemplo:

```
val = max(10,20); // agora, val é 20
```

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

Exemplo:

```
val = abs(-5);   // agora, val é 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 maiores
                                        // 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 208
```

`double pow(base expoente)` - Retorna o resultado da potenciação de um número (base) por i valor específico (expoente).

Exemplo:

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

Exemplo:

```
double tangent = tan(2); // aproximadamente -2,18503975868
```

Funções de números aleatórios

Caso você tenha de gerar números aleatórios, pode utilizar o gerador de números pseudoaleatórios rduino.

`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, você deve redefinir o gerador utilizando um valor aleatório. Caso você tenha um pino analógico conectado, 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 o ruído do pino 5
```

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

Exemplo:

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

Comunicação serial

Como vimos no capítulo 5, você pode efetuar a comunicação com dispositivos por meio da porta USB utilizando um protocolo de comunicação serial. Veja, 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, você geralmente utilizará 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(9609);
```

`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(1S, 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 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.