Universidade Federal Fluminense - UFF
Escola de Engenharia - TCE
Curso de Engenharia de Telecomunicações - TGT

Programa de Educação Tutorial - PET
Grupo PET-Tele

Tópicos Especiais em Eletrônica II

Introdução ao microcontrolador Arduino

Apostila de programação

(Versão: A2014M05D02)

Autor: Roberto Brauer Di Renna
Lorraine de Miranda Paiva
Tutor: Alexandre Santos de la Vega

Niterói - RJ
Fevereiro / 2014

Sumário

Lista de Figuras

Capítulo 1

Estrutura

1.1 setup()

A função `setup()` é chamada quando um programa começa a rodar. É usada para inicializar as variáveis, os tipo dos pinos, declarar o uso de bibliotecas, entre outros. Esta função será executada apenas uma vez após a placa Arduino ser ligada ou reiniciada.

Exemplo

```
int buttonPin = 3;

void setup(){

  Serial.begin(9600);
  pinMode(buttonPin, INPUT);
}

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

1.2 loop()

Após criar uma função `setup()` que declara os valores iniciais, a função `loop()` faz exatamente o que seu nome sugere, entra em *looping* (executa sempre o mesmo bloco de código), permitindo ao seu programa fazer mudanças e responder. Esta função é usada para controlar ativamente a placa Arduino.

Exemplo

```
int buttonPin = 3;
```

```
// setup inicializa o serial e o pino do button (botao)
void setup(){

  beginSerial(9600);
  pinMode(buttonPin, INPUT);
}

// loop checa o botao a cada vez,
// e envia o serial se ele for pressionado
void loop(){

  if (digitalRead(buttonPin) == HIGH)
    serialWrite('H');
  else
    serialWrite('L');

  delay(1000);
}
```

1.3 Estruturas de controle

1.3.1 if e operadores de comparação

Essa estrutura, que é usada em conjunto com um operador de comparação, testa alguma condição imposta pelo programador. Para testar se o valor de uma variável é maior do que 100 por exemplo,

```
if (variavel > 100){

  // faça algo aqui
}
```

O programa testa se `variavel` é maior do que 100. Se for, o programa executa a ação seguinte. Caso contrário, o programa ignora a ação dentro das chaves e segue a rotina. Caso a ação tenha apenas uma linha, as chaves podem ser omitidas.

Para fazer as comparações, é necessária a utlilização dos operadores da Tabela1.3

Tabela 1.1: Operadores de comparação.

Operador	Operação
==	Igual a
!=	Diferente de
<	Menor que
>	Maior que
<=	Menor ou igual a
>=	Maior ou igual a

1.3.2 if / else

A estrutura `if/else` fornece um controle maior sobre o código do que simplesmente o `if`, permitindo múltiplos testes agrupados. Por exemplo, uma entrada analógica poderia se testada e uma ação executada caso o valor fosse menor do que 500. Caso seja maior ou igual a 500, executaria a segunda ação.

```
if (analog < 500){
  // ação A
}
else{
  // ação B
}
```

Também é possível colocar uma condição logo após do `else`. Não há limites de condições a serem colocadas.

```
if (analog < 500)
{
  // ação A
}
else if (analog >= 1000)
{
  // ação B
}
else
{
  // ação C
}
```

Outra forma de se fazer condições é com a estrutura da subsecção 1.3.4, o `swich case`.

1.3.3 for

`For` é chamada de estrutura de laço, pois cada bloco de programa se repete uma determinada quantidade de vezes. Um contador é normalmente usado para terminar e dar ritmo ao *loop*. É útil para qualquer operação que se repita. Uma aplicação comum é operações com vetores.

Sintaxe:

```
for (inicialização; condição; incremento) {
//ação;
}
```

A inicialização começa pela primeira e única vez. Cada vez que iniciará um novo *loop*, a condição é testada. Se for verdadeira, a ação é executada e a variável de passo incrementada. Quando a condição se tornar falsa, o *loop* termina.

Exemplo

```
// Escurecer um LED usando um pino a PWM
int PWMpin = 10; // LED em série com um resistor de 470 ohm no pino 10

void setup(){
  // não há necessidade de setup
}

void loop(){
   for (int i=0; i <= 255; i++){
      analogWrite(PWMpin, i);
      delay(10);
   }
}
```

Dica de código

C é uma linguagem que possibilita ao usuário maior conforto do que outras linguagens para realizar determinadas ações. Uma delas é o *loop*. É possível que, para inicialização, condição, e incremento podem ser quaisquer declarações válidas em C e fazer uso de variáveis ??independentes, além de utilizar quaisquer tipos de dados, incluindo o tipo *float*. Isso pode fornecer soluções para alguns problemas complicados de programação.

Por exemplo, multiplicar o incremento gerando uma progressão logarítmica:

```
for(int x = 2; x < 100; x = x * 1.5){
println(x);
}
```

Gera: 2,3,4,6,9,13,19,28,42,63,94.
Outro exemplo, enfraquecer e aumentar a luz de um LED:

```
void loop(){
   int x = 1;
   for (int i = 0; i > -1; i = i + x){
      analogWrite(PWMpin, i);
      if (i == 255) x = -1;             //troca a direção do pico
      delay(10);
   }
}
```

1.3.4 switch / case

Como a estrutura `if`, `switch...case` controla os programas permitindo os programadores especificar diferentes código que poderão ser executados em diferentes condições. Em particular, a estrutura do `switch` compara o valor da variável com o valor em cada caso específico. Quando o caso é encontrado, ou seja, o valor é o da variável, a ação correspondente é executada.

A palavra reservada `break` interrompe o `switch`, e é típicamente usado no final de cada `case`. Sem o `break`, a estrutura do `switch` irá ser executada até encontrar uma interrupção, ou o final das comparações.

Example

```
  switch (var) {
    case 1:
      //faça algo quando var equivale a 1
      break;
    case 2:
      //faça algo quando var equivale a 2
      break;
    default:
      // se nenhuma comparação for verdadeira, faz-se o padrão
      // o padrão pode ou não, existir
  }
```

Sintaxe

```
switch (var) {
  case label:
    // ação
    break;
  case label:
    // ação
    break;
  default:
    // ação
}
```

Parâmetros
var: a variável que será comparada
label: o valor que será comparado com a variável "var"

1.3.5 while

O laço do `while` acontece infinitamente até que a expressão dentro dos parênteses se torne falsa. Algo deve ser modificado ao testar a variável para o término do *loop*, caso contrário ele nunca terminará.

Sintaxe

```
while(expressão){
  // ação
}
```

Parâmetros

expressão - A estrutura (booleana)de C que verifica se é verdadeiro ou falso.

Exemplo

```
var = 0;
while(var < 200){
  // faça algo duzentas vezes
  var++;
}
```

1.3.6 do-while

Muito parecido com o `while`, o do-while possui a diferença que sua condição é testada ao final do *loop*, assim executando a ação garantidamente uma vez.

Sintaxe

```
do{
    // bloco de ações
} while (teste de condição);
```

Exemplo

Leitura de algum sensor:

```
do{
  delay(50);          // esperar sensores estabilizarem
  x = readSensors();  // leitura de sensores. fará 100 leituras

} while (x < 100);
```

1.3.7 break

Interrompe alguma condição de *loop*. Também é usado na estrutura de `switch`.

Exemplo

```
for (x = 0; x < 255; x ++){
    digitalWrite(PWMpin, x);
    sens = analogRead(sensorPin);
    if (sens > threshold){
       x = 0;
       break;
    }
    delay(50);
}
```

1.3.8 continue

Ignora o resto da iteração atual de um *loop*. `continue` continua marcando a expressão condicional do *loop*, e prossegue com as iterações subsequentes.

Exemplo

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

    digitalWrite(PWMpin, x);
    delay(50);
}
```

1.3.9 return

Termina a função e retorna um valor de uma função para a função de chamada.

Sintaxe

```
return;
return valor; // ambas as formas são válidas
```

Parâmetros

valor: qualquer variável ou constante, de qualquer tipo.

Exemplos:

```
 int checkSensor(){
    if (analogRead(0) > 400) {
        return 1;
    else{
        return 0;
    }
}
```

Dica de código

`return` é útil para testar partes de código ao invés de fazer um bloco de comentário.

```
void loop(){

// Ideia brilhante de código aqui

return;

// resto do código inútil
// essa parte nunca será executada
}
```

1.3.10 goto

Transfere o fluxo do programa para um ponto específico do mesmo programa.

Sintaxe

```
label:
goto label; // envia o fluxo do programa para o label
```

Dica de código

Alguns autores desencorajam o uso do `goto` em C alegando nunca ser necessário, no entanto, pode simplificar certos programas. Uma dessas situações é de encerrar longos *loops* ou blocos lógicos, com uma determinada condição./ A razão pela qual muitos programadores desaprovam, é que com o uso desenfreado de instruções `goto`, é fácil criar um programa com o fluxo do programa indefinido, o que nunca pode ser depurado.

Exemplo

```
for(byte r = 0; r < 255; r++){
    for(byte g = 255; g > -1; g--){
        for(byte b = 0; b < 255; b++){
            if (analogRead(0) > 250){ goto bailout;}
        }
    }
}
bailout:
```

1.4 Elementos de sintaxe

1.4.1 ; - ponto e vírgula

Usado ao terminar uma linha.

Exemplo

```
int a = 13;
```

Dicas de código

Terminar uma linha sem o ponto e vírgula resultará em um erro de compilação. Uma das primeiras coisas a se verificar quando há algum erro sem razão, é um ponto e vírgula esquecido, que precede a linha em que o compilador reclamou.

1.4.2 {} - Chaves

As chaves são largamente usadas na programação em C. Possuem diversas aplicações, descritas abaixo, que podem causar confusão em iniciantes.

Toda chave de abertura ({) deve ser fechada (}). Essa é a condição que se refere às chaves estarem balanceadas. A IDE (*Integrated Development Enviroment*, em inglês) do Arduino inclui um artifício que confere se as chaves estão devidamente balanceadas. Para isso, deve-se apenas selecionar uma chave que a correspondente irá ser destacada.
Atualmente, esse artifício possui um pequeno *bug* onde ele também destaca (incorretamente) as chaves que estão comentadas. Programadores iniciantes e programadores que habituados com BASIC normalmente encontram dificuldades com o uso de chaves. Não obstante, as chaves substituem a instrução `RETURN` em uma subrotina (função), a estrutura `ENDIF` por uma condicional e a instrução `Next` por um *loop* `for`.
Por conta do uso das chaves ser bem variado, uma dica para praticar é imediatamente após se abrir uma chave é fechá-la. Chaves desbalanceadas causam muitos erros de compilação. Muitas vezes fica complicado encontrar o erro de sintaxe em um código de muitas linhas. Por conta de seu largo uso, as chaves são muito importantes para a sintaxe do programa. Movendo uma ou duas linhas de lugar, o sentido do código pode-se alterar drasticamente.

Funções

```
void minhafunção(parâmetro){
  ações
}
```

Loops

```
while (expressão booleana)  {
   ações
}

do  {
```

```
    ações
} while (expressão booleana);

for (incialização; condição de término; expressão de incremento) {
    ações
}
```

Estruturas condicionais

```
if (expressão booleana) {
    ações
}

else if (expressão booleana) {
    ações
}
else {
    ações
}
```

1.4.3 // e /∗ - Comentários

Comentários são linhas de programa que são usadas para informar o próprio programador ou outra pessoa que venha a usar o código como o programa funciona. Essas linhas são ignoradas pelo compilador e nem são enviadas para o processador, logo, não ocupam espaço no *chip* do Atmega. Comentários possuem o propósito de ajudar o programador a entender (ou lembrar) como o programa funciona ou para informar a outros. Existem duas formas diferentes de comentar:

Example

```
x = 5;  // Essa é uma linha simples de comentário.
        // Tudo escrito aqui não será considerado
         // outra linha de comentário

/* a barra seguida do asterisco indica um bloco de comentário.
até um asterisco seguido de barra, nada será considerado.

if (gwb == 0){   // linha única de comentário dentro de um comentário maior.
x = 3;
}

*/ fecha o bloco de comentário
```

É importante destacar que não se pode fazer um bloco de comentário dentro de outro.

Dica de código

Ao programar, uma técnica para se testar partes de código é comentar trechos ao invés de apagar linhas. Assim, o compilador ignora esses trechos e o programador pode verificar como fica o programa nessa nova configuração, sem perder o que já foi digitado.

1.4.4 define

`define` permite ao programador nomear um valor de uma constante antes do programa ser compilado. Constantes nomedadas por `define` não usam memória do *chip*. O compilador irá fazer referências para essas constantes com o valor definido no momento da compilação.
Isso pode causar alguns efeitos indesejáveis como o nome definido por `define` ser incluído em outra constante ou nome de variável.Nesse caso o texto deverá ser trocado pelo número ou texto que se usou o `define`.
Em geral, a palavra reservada `const` é melhor para se definir constantes do que o `define`.

A sintaxe do Arduino é a mesma que em C:

Sintaxe

```
#define nomedaconstante valor
```

Note que o # é necessário.

Exemplo

```
#define ledPin 3
// O compilador irá trocar ledPin por 3 no momento de compilar.
```

Dica de código

Não há ponto e vírgula na estrutura do `define`. Caso colocar algum, o compilador apontará erros.

```
#define ledPin 3;    // errado!
```

Colocar um sinal de igual também é um erro de sintaxe.

```
#define ledPin  = 3  // errado!
```

1.4.5 include

`#include` é usado para incluir bibliotecas. Possibilita ao programador usar um largo grupo de funções pré-definidas. Existem bibliotecas escritas especificamente para Arduino.
A página de referência de bibliotecas de C da AVR(AVR é a referência dos *chips* da Atmel, os quais o Arduino é baseado) pode ser conferido neste link[1].
Note que o `include` é similar ao `define`, não há ponto e vírgula para terminar a linha, causando assim um erro de sintaxe caso seja colocado.

Exemplo

Esse exemplo inclui a biblioteca que é usada para colocar informações no espaço de memória *flash* ao invés da *ram*. Isso guarda espaço da *ram* para a mamória dinâmica e faz a procura por tabelas mais prática.

```
#include <avr/pgmspace.h>

prog_uint16_t myConstants[] PROGMEM = {0, 21140, 702   , 9128,  0, 25764, 8456,
0,0,0,0,0,0,0,0,29810,8968,29762,29762,4500};
```

1.5 Operadores aritméticos

Na Tabela 1.2 podemos conferí-los:

Tabela 1.2: Operadores aritméticos.

Operador	Operação
=	Atribuição
+	Adição
−	Subtração
*	Multipicação
/	Divisão
%	Resto da divisão inteira

[1] http://www.nongnu.org/avr-libc/user-manual/modules.html+

1.6 Operadores de comparação

Na Tabela 1.3 podemos conferí-los:

Tabela 1.3: Operadores de comparação.

Operador	Operação
==	Igual a
!=	Diferente de
<	Menor que
>	Maior que
<=	Menor ou igual a
>=	Maior ou igual a

1.7 Operadores Booleanos

Na Tabela 1.4 podemos conferí-los:

Tabela 1.4: Operadores booleanos.

Operador	Significado lógico
&&	e (and)
||	ou (or)
!	negação (not)

1.8 Operadores de ponteiro

Na Tabela 1.5 podemos conferí-los:

Tabela 1.5: Operadores de ponteiro.

Operador	Significado
&	Referencia
*	Desreferencia

Ponteiros são um dos pontos mais complicados para os inciantes em linguagem de programação C e é possível escrever a maioria dos códigos de Arduino sem usar ponteiros. Entretanto manipulando certas estruturas de dados, o uso de ponteiros simplificam o código e o conhecimento de como manipulá-los é uma ferramenta importante para o programador de C.

1.9 Operadores bit a bit (*Bitwise*)

1.9.1 *Bitwise* E (AND) (&), *Bitwise* OU (OR) (|), *Bitwise* OU EXCLUSIVO (XOR) (^)

Os operadores bit a bit, como o nome sugere, baseiam-se seus cálculos a nível de bit das variáveis. Esses operadores resolvem uma larga quantidade de problemas.

Abaixo seguem descrições e sintaxe de todos os operadores. Para mais detalhes, verifique a bibliografia[3].

Bitwise AND (&)

O operador *bitwise AND* é equivalente ao operador lógico "AND".
O operador *bitwise AND* representado por "&" é usado entre duas expressões inteiras. *Bitwise AND* opera em cada bit de cada expressão independentemente, seguindo a seguinte regra: se ambos são 1, o resultado é 1, qualquer outro caso, a resposta é 0. A tabela-verdade da porta lógica AND pode ser conferida na Tabela 1.6, abaixo:

Tabela 1.6: Tabela-verdade porta AND.

A	B	Resultado
0	0	0
1	0	0
0	1	0
1	1	1

Outra forma de expressar isso:

```
0  0  1  1    operando 1
0  1  0  1    operando 2
----------
0  0  0  1    (operando 1 & operando 2) - resultado retornado
```

No Arduino, o tipo inteiro (int) é um valor de 16 bits, então ao usar o & entre duas expressões causa 16 operações simultâneas. Um exemplo de código:

```
int a =  92;    // em binário: 0000000001011100
int b = 101;    // em binário: 0000000001100101
int c = a & b;  // resultado:  0000000001000100, ou 68 em decimal.
```

Cada um dos 16 bits em "a" e "b" são processados usando o *bitwise AND*, e todos os 16 bits resultantes são guardados em "c", resultando no valor de 01000100 em binário, equivalente a 68 em decimal.

Bitwise OR (|)

O operador *bitwise OR* é equivalente ao operador lógico "OR".
O operador *bitwise OR* representado por "|" é usado entre duas expressões inteiras. *Bitwise OR* opera em cada bit de cada expressão independentemente, seguindo a seguinte regra: se ambos são 0, o resultado é 0, qualquer outro caso, a resposta é 1. A tabela-verdade da porta lógica OR pode ser conferida na Tabela 1.7, abaixo:

Tabela 1.7: Tabela-verdade porta OR.

A	B	Resultado
0	0	0
1	0	1
0	1	1
1	1	1

Outra forma de representar:

```
0  0  1  1    operando 1
0  1  0  1    operando 2
----------
0  1  1  1    (operando 1 | operando 2) - resultado retornado
```

Um exemplo de código:

```
int a = 92;      // binário: 0000000001011100
int b = 101;     // binário: 0000000001100101
int c = a | b;   // binário: 0000000001111101, ou 125 em decimal.
```

Bitwise XOR (^)

Há também um operador não tão usado em C, chamado de "ou-exclusivo", também conhecido como XOR. O operador *bitwise XOR* representado por "^" é usado entre duas expressões inteiras. *Bitwise XOR* opera em cada bit de cada expressão independentemente. Sua regra é muito parecida com operador OR, trocando apenas o caso em que temos 1 e 1, onde o resultado é 0. A tabela-verdade da porta lógica OXR pode ser conferida na Tabela 1.8, abaixo:

Tabela 1.8: Tabela-verdade porta XOR.

A	B	Resultado
0	0	0
1	0	1
0	1	1
1	1	0

Outra forma de representar:

```
0  0  1  1    operando 1
0  1  0  1    operando 2
----------
0  1  1  0    (operando 1 ^ operando 2) - resultado retornado
```

Exemplo de código:

```
int x = 12;      // binário: 1100
int y = 10;      // binário: 1010
int z = x ^ y;   // binário: 0110, ou decimal 6
```

Dica de código

O operador XOR também é usado como um somador de números binários.

1.9.2 *Bitwise* NOT(∼)

O operador *bitwise* NOT, é representado por "∼". Diferentemente de & e |,
sim é aplicado a apenas um operando. NOT troca cada bit por seu oposto: 0 torna-se 1 e 1 torna-se 0. A tabela-verdade da porta lógica OXR pode ser conferida na Tabela 1.9, abaixo:

Tabela 1.9: Tabela-verdade porta NOT.

A	Resultado
0	1
1	0

Outra forma de representar:

```
 0  1        operando 1
------
 1  0        NOT operando 1

 int a = 103;    // binário:  0000000001100111
 int b = NOT a;     // binário:  1111111110011000 = -104
```

Não está incorreto o resultado ser −104. Isto é porque o bit mais elevado em uma variável inteira é o chamado bit de sinal. Se o bit mais alto é 1, o número é interpretado como negativo. Esta codificação de números positivos e negativos é chamado de complemento de dois. Para mais informações, consulte a bibliografia.
Note que para um inteiro x qualquer, ∼ equivale a $-x-1$.

1.9.3 *Bitshift left* (<<), *Bitshift right* (>>)

Existem dois operadores de deslocamento bit em C: o operador de deslocamento à esquerda << e o operador de deslocamento à direita >>. Estes operadores fazem com que os bits do operando esquerdo sejam deslocados para a esquerda ou para a direita pelo número de posições especificado pelo operando direito.

Sintaxe

variável << `numero_de_bits`
variável >> `numero_de_bits`

Parâmetros
variável - (byte, int, long) `numero_de_bits` integer <= 32

Exemplo:

```
int a = 5;        // binário: 0000000000000101
int b = a << 3;   // binário: 0000000000101000, ou 40 em decimal
int c = b >> 3;   // binário: 0000000000000101, ou volta para o 5 como iniciou
```

Ao mudar um valor x por bits y (x << y), os bits de y mais à esquerda são perdidos em x, literalmente deslocado para fora do conjunto:

```
int a = 5;        // binário: 0000000000000101
int b = a << 14;  // binário: 0100000000000000 - o primeiro 1 do 101 foi descartado
```

Se você tiver certeza de que nenhum dos números significativos em um valor estão sendo deslocados para "fora", uma maneira simples de pensar no operador de deslocamento á esquerda é que ele multiplica o operando da esquerda por 2 elevado á potência operando á direita. Por exemplo, para gerar potências de 2, as seguintes expressões podem ser empregadas:

```
1 <<  0  ==    1
1 <<  1  ==    2
1 <<  2  ==    4
1 <<  3  ==    8
...
1 <<  8  ==  256
1 <<  9  ==  512
1 << 10  == 1024
...
```

Quando você muda x direita por bits y (x » y), e o bit mais alto em x é um 1, o comportamento depende do tipo de dados exatos de x. Se x é do tipo inteiro, o bit mais alto é o bit de sinal, que determina se x é negativo ou não, como já discutido anteriormente. Nesse caso, o bit de sinal é copiado para os bits mais baixos:

```
int x = -16;     // binário: 1111111111110000
int y = x >> 3;  // binário: 1111111111111110
```

Esse comportamento, chamado de extensão de sinal, muitas vezes não é o comportamento desejado. Em vez disso, pode-se desejar zeros a serem deslocados para a esquerda. Acontece que as regras deslocamento para a direita são diferentes para expressões do tipo inteira sem sinal, então a solução é usar um *typecast* para suprimir os bits que estão sendo copiados da esquerda:

```
int x = -16;                    // binário: 1111111111110000
int y = (unsigned int)x >> 3;   // binário: 0001111111111110
```

Sendo cuidadoso para evitar a extensão de sinal, é possível pode usar o operador de deslocamento para a direita >> como uma forma de dividir por potências de 2. Por exemplo:

```
int x = 1000;
int y = x >> 3;   // divisão inteira de 1000 por 8, tendo como resultado y = 125.
```

1.10 Operadores de composição

1.10.1 Incremento e decremento

Incrementa (++) ou decrementa uma variável.

Sintaxe

```
x++;  // incrementa x em um e retorna o valor anterior de x
++x;  // incrementa x em um e retorna o novo valor de x

x--;  // decrementa x em um e retorna o valor anterior de x
--x;  // decrementa x em um e retorna o novo valor de x
```

Parâmetros

x pode ser determinado como "integer" ou "long".

Retorna

O valor original ou o valor incrementado/decrementado de uma variável.

Exemplo

```
x = 2;
y = ++x;      // x agora possui o valor de 3 e y tambem 3
y = x--;      // x possui o valor de 2 novamente e y continua com o valor de 3
```

1.10.2 Operações compostas

São basicamente simplificações para as operações matemáticas. A Tabela 1.10 lista as possibilidades:

Tabela 1.10: Operações compostas.

Forma simplificada	Equivalência usual
$x += y;$	$x = x + y$
$x -= y;$	$x = x - y$
$x *= y;$	$x = x * y$
$x /= y;$	$x = x/y$
$x \&= y$	$x = x\&y$
$x \mid= y$	$x = x\|y$

É importante destacar que para a forma composta do *bitwise AND* e do *bitwise OR*, o "x" pode ser do tipo caractere, inteiro(int) ou longo (long). "y" possui a liberdade de ser uma constante inteira de qualquer um dos tipos citados. Para as outras operações, pode-se usar qualquer tipo de variável.

Exemplo

```
x = 2;
x += 4;      // x agora possui o valor de 6
x -= 3;      // x agora possui o valor de 3
x *= 10;     // x agora possui o valor de 30
x /= 2;      // x agora possui o valor de 15
```

Capítulo 2

Variáveis

2.1 Constantes

2.1.1 true | false

false

`false` é definida como 0 (zero).

true

`true` é normalmente definida como 1, o que está certo, apesar de possuir uma definição mais larga. Qualquer inteiro diferente de zero é `true`, no pensamento Booleano. Então, pode-se chegar a conclusão que -1, 2 e -200 são todos definidos como `true` também.

Dica de código

Repare que diferentemente das palavras reservadas `HIGH`, `LOW`, `INPUT` e `OUTPUT`, `true` e `false` são escritas em letras minúsculas.

2.1.2 HIGH | LOW

No momento que se for ler ou escrever em um pino digital, há apenas dois valores que o pino pode assumir: `HIGH` ou `LOW`.

HIGH

Em relação aos pinos, o significado de `HIGH` é diferente dependendo de qual modo o pino for colocado, se `INPUT` ou `OUTPUT`. Quando o pino é configurado como `INPUT` com a função `pinMode` e lido com a função `digitalRead`, o microcontrolador irá considerar como `HIGH` se a tensão for maior ou igual a 3 volts no pino.

Também é o possível configurar com `pinMode` o pino como `INPUT` e, posteriormente, com a leitura do `digitalWrite` como `HIGH` definirá os 20K resistores *pull-up* internos que orientarão o pino de entrada para o valor de `HIGH` exceto no caso que for puxado para `LOW` pelo circuito

externo. É dessa forma que a função `INPUT_PULLUP` funciona.

Quando um pino é configurado como `OUTPUT` com `pinMode`, e definido como `HIGH` com `digitalWrite`, o pino está com 5 volts. Neste estado, é possível fornecer corrente, por exemplo, acender um LED que está conectado através de um resistor em série para o terra ou para o outro pino configurado como `OUTPUT`, e definido como `LOW`.

LOW

O significado de `LOW` também difere dependendo de como o pino foi definido. Quando configurado como `INPUT` com `pinMode`, e lido com `digitalRead`, o microcontrolador irá considerar como `HIGH` se a tensão for menor ou igual a 2 volts no pino.

Quando um pino é configurado como `OUTPUT` com `pinMode`, e definido como `LOW` com `digitalWrite`, o pino está com 0 volts. Neste estado, é possível reduzir corrente, por exemplo, acender um LED que está conectado através de um resistor em série para Vcc, ou para o outro pino configurado como `OUTPUT`, e definido como `HIGH`.

2.1.3 INPUT | OUTPUT | INPUT_PULLUP

Pinos digitais podem ser usados como `INPUT`, `INPUT_PULLUP`, ou `OUTPUT`. Modificar o pino com `pinMode()` singnifica modificar o comportamento elétrico do pino.

Pinos configurados como INPUT

Os pinos do Arduino definidos como `INPUT` com `pinMode()` estão em estado de alta-impedância. Pinos configurados como `INPUT` possuem capacidade apenas para executar ações de pequeno porte. Equivalem a uma resistência em série de 100MΩ na frente do pino. Isso os torna úteis para a leitura de um sensor, mas não alimentar um LED.

Pinos configurados como INPUT_PULLUP

O chip interno Atmega do Arduino possui resistores do tipo *pull-up*[1] internos que podem ser acessados. Caso seja necessários usar esses resistores ao invés de resistores externos *pull-down*[2], deve-se usar `INPUT_PULLUP` como argumento de `pinMode()`. Isso efetivamente inverte o comportamento, onde `HIGH`significaria que o sensor está desligado e `LOW`, ligado.

Para mais informações sobre resitores *pull-up* e *pull-down*, veja o link[3].

[1] A idéia de um resistor *pull-up* é que ele fracamente "puxe(*pulls*)" a tensão do condutor que ele está conectado para 5V (ou qualquer tensão que represente o nível lógico "alto").

[2] resistores *pull-down* são usados para armazenar a entrada em valor zero(baixo) quando nenhum outro componente estiver conduzindo a entrada. Eles são usados com menos frequência que os resistores *pull-up*.

[3] `http://pt.wikipedia.org/wiki/Resistores`$_{p}ull - up+$

Pinos configurados como Output

Pinos configurados como `OUTPUT` com `pinMode()` estão em estado de baixa-impedância. Podem prover uma quantidade considerável de corrente para outros circuitos. Pinos da Atmega fornecem corrente possitiva ou negativa até $40mA$ (miliampères) de corrente para outros circuitos. Isso é útil para acender LEDs, mas não para ler sensores. Pinos definidos como `OUTPUT` podem ser destruídos caso sejam submetidos a curto-circuito com 5 volts. A quantidade de corrente fornecida por um pino Atmega também não é suficiente para abastecer a maioria dos relés ou motores, e alguns circuitos de interface serão necessários.

2.1.4 LED_BUILTIN

A maior parte das placas Arduino possuem um pino conectado a um LED *onboard* em série com um resistor. `LED_BUILTIN` é uma forma de declarar manualmente o pino como uma variável. A maior parte das placas possuem esse LED conectado ao pino digital 13.

2.1.5 Constantes inteiras

Consatantes inteiras são números usados diretamente no *sketch*, como 123. Por definição, esses números são tratados como "int", mas é possível modificar isso com os modificadores U e L (veja abaixo).
Normalmente, constantes inteiras são tratadas na base 10 (decimal), mas notações especiais podem ser usadas como entrada de números em outras bases.

Tabela 2.1: Constantes inteiras.

Base	Exemplo	Formatador	Comentário
10 (decimal)	123	nenhum	
2 (binário)	B1111011	condutor "B"	funcionam apenas valores de 8 bits (0 a 255) caracteres 0 e 1 são válidos
8 (octal)	0173	condutor "0"	caracteres de "0" a "7" são válidos
16 (hexadecimal)	0x7B	condutor "0x"	caracteres "0" a "9", "A" a "F", "a" a "f" válidos

Base decimal
Decimal é base 10. Constantes sem prefixos são assumidas como formato decimal.
Exemplo:

```
101     // mesmo que o 101 na base decimal   ((1 * 10^2) + (0 * 10^1) + 1)
```

Base binária
Binário é base dois. Apenas caracteres 0 e 1 são válidos.

Exemplo:

```
B101    // mesmo que 5 na base decimal   ((1 * 2^2) + (0 * 2^1) + 1)
```

O formatador binário apenas trabalha com *bytes* (8 *bits*) entre zero (B0) e 255 (B11111111). Se for conveniente inserir um inteiro (int, 16 bits) na forma binária, deve-se fazer em dois passos:

```
+myInt = (B11001100 * 256) + B10101010;
```

Base octal

Octal é a base oito. Apenas caracteres de 0 a 7 são válidos. Valores em octal são indicados com o prefixo "0".

Exemplo:

```
0101    // mesmo que 65, em decimal   ((1 * 8^2) + (0 * 8^1) + 1)
```

Dica de código

É possível acontecer um erro difícil de se encontrar incluindo um zero na frente da constante, o compilador interpreta a constante como octal.

Hexadecimal

Hexadecimal (ou hex) é a base dezesseis. Caracteres válidos são de 0 a 9 e letras de "A" a "F". "A" possui o valor de 10, "B" de 11, até "F", que vale 15. Valores Hex são indicados com o prefixo "0x". Repare que tanto faz escrever as letras em caixa alta ou baixa.

Exemplo:

```
0x101   // mesmo de 257, em decimal   ((1 * 16^2) + (0 * 16^1) + 1)
```

Formatadores U e L

Por definição,, uma constante inteira é tratada como um inteiro com limitações inerentes a seus valores possíveis. Para especificar uma constante inteira com outros tipos de dados:

"u" ou "U" para forçar a constante ao formato de dados "unsigned". Exemplo: 33u
"l" ou "L" para forçar a constante ao formato de dados "long data". Exemplo: 100000L
"ul" ou "UL" para forçar a constante ao formato de dados "unsigned long constant". Exemplo: 32767ul

2.1.6 Constantes de ponto flutuante

Similar a constantes inteiras, contantes de ponto flutuante são usadas para tornar o código mais legível. Constantes de ponto futuante são trocados na compilação para o valor para o qual a expressão é avaliada.

Exemplo:

```
n = .005;
```

Constantes de ponto flutuante também podem ser expressas em uma variação de notação científica. “E” e “e” são ambas aceitas como expoentes válidos.

Tabela 2.2: Constantes de ponto flutuante.

Ponto flutuante constante	equivale a:	assim como a:
10.0	10	
2.34E5	$2.34 * 10^5$	234000
67e-12	$67.0 * 10^-12$	.000000000067

2.2 Tipos de dados

2.2.1 void

A palavra reservada `void` é usada apenas no momento de declarar a função. Indica que a função é esperada para retornar a informação para a função que foi chamada.

Exemplo:

```
// ações acontecem nas funções "setup" e "loop"
// mas nenhuma informação foi repassada para a maior parte do programa

void setup(){
  // ...
}

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

2.2.2 boolean

Uma variável do tipo booleana assume dois valores, de *true* ou *false*. Cada variável deste tipo ocupa um byte de memória.

2.2.3 char

É um tipo de dado que utiliza apenas um byte de memória para gurdar o valor de um caractere. Para um caractere usamos aspas simples `'A'`, por exemplo, e para múltiplos caracteres(*strings*) aspas duplas: `"ABC"`.
Caracteres são armazenados como números, entretanto. Percbe-se pela codificação ASCII(*American Standard Code for Information Interchange*, em inglês). Isso significa que é possível fazer contas com caracteres, usando os valores da tabela ASCII (ex. 'A' + 1 possui o valor de 66, já que na tabela ASCII a letra a maiúscula possui o valor de is 65). Veja mais valores na Figura. A.1
O tipo de dados `char` é um tipo *signed*, o que significa que ele codifica os números de -128 a 127. Para um, de um *byte* (8 *bits*) tipo de dados não *signed*, use o tipo de dados *byte*.

Exemplo

```
char myChar = 'A';
char myChar = 65;      // ambos são válidos
```

2.2.4 unsigned char

É um tipo de dado que ocupa apenas 1 *byte* de memória. O tipo de dado aceita de 0 a 255.
No estilo de programação no Arduino, o tipo de dados *byte* deve ser preferido.

Exemplo

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

2.2.5 byte

A *byte* guarda um número de 8 bits, de 0 a 255.

Example

```
byte b = B10010;  // "B" \'{e} o formatador bin\'{a}rio (B10010 = 18 em decimal)
```

2.2.6 int

Inteiro é a primeira forma com que se inicia a guardar números.
No Arduino Uno (e outras placas da ATMega) uma variável do tipo inteiro (int) guarda um valor de 16 *bits* (2 *bytes*). Isso permite um intervalo de $-32,768$ a $32,767$ (valor mínimo de -2^15 e valor máximo de (2^{15}-1)).
Já no Arduino Due, `int` guarda valores de 32 *bits* (4 *bytes*). Isso permite um intervalo de $-2,147,483,648$ a $2,147,483,647$ (valor mínimo de -2^{31} e valor máximo de (2^{31}) - 1).

`int` guarda números negativos com uma técnica chamada de complemento de dois, citada anteriormente. O *bit* mais significativo, algumas vezes chamado de *bit* de sinal, torna o número negativo. O Arduino "se preocupa" em lidar com números negativos logo, as operações aritméticas ocorrem de maneira esperada. No entanto, podem acontecer alguns problemas ao lidar com o operador *bitshift right* (>>).

Exemplo

```
int ledPin = 13;
```

Sintaxe

```
int var = val;
```

```
var - nome da variável inteira
val - o valor que a variável assumirá
```

Dica de código

Quando variáveis excedem sua capacidade máxima de armazenamento, elas "rolam" para sua capacidade mínima. Note que isso ocorre em todas as direções.

Exemplo para uma variável inteira de 16 bits:

```
int x;
x = -32768;
x = x - 1;       // x agora guarda 32,767 - rola para a direção negativa

x = 32767;
x = x + 1;       // x possui o valor de -32,768 - rola para a direção positiva
```

2.2.7 unsigned int

No Arduino UNO e outras placas da ATMEGA, `unsigned ints` (inteiros *unsigned*) são o mesmo das inteiras que guardam valores de 2 *bytes*. Ao invés de guardar números negativos apenas guarda valores positivos, em um intervalo de 0 a $65.535(2^{16}) - 1$). A versão Due guarda valores de 4 *bytes* (32 *bits*), variando de 0 a $4,294,967,295$ ($2^{32} - 1$).
A diferença entre `unsigned int` e (*signed*) `int`, é que o MSB[4] é considerado de sinal, assim considerando o número negativo, mais uma vez usando o conceito do complemento de 2.

Exemplo

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

[4] *Most Significant Bit*, *Bit* mais significativo, em inglês. Verifique a bibliografia[3] para mais informações.

Sintaxe

```
unsigned int var = val;

var - nome da variável
val - valor assumido pela vairável
```

Dica de código

Quando variáveis excedem sua capacidade máxima de armazenamento, elas "rolam" para sua capacidade mínima. Note que isso ocorre em todas as direções.

```
unsigned int x
x = 0;
x = x - 1;       // x agora guarda 65535 - rola para a direção negativa
x = x + 1;       // x possui o valor de 0 - rola para a direção positiva
```

2.2.8 word

Armazena um número do tipo *unsigned* de 16 *bits* de 0 a 65535. Semelhante a `unsigned int`.

Exemplo

```
word w = 10000;
```

2.2.9 long

Converte o valor para o tipo `long`.

Sintaxe

```
long(x)
```

Parâmetros

```
x: varável de qualquer tipo
```

Retorna

```
long
```

2.2.10 unsigned long

Variáveis do tipo `unsigned long` possuem tamanhos extensos em relação ao armazenamento de números e podem guardar 32 *bits* (4 *bytes*). `unsigned long` não guarda números negativos, tendo como intervalo de 0 a $4,294,967,295 (2^{32} - 1)$.

Exemplo

```
unsigned long time;

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

void loop(){
  Serial.print("Time: ");
  time = millis();
  //escreve desde que o programa tenha iniciado
  Serial.println(time);
  delay(1000);
}
```

Sintaxe

```
unsigned long var = val;
```

```
onde:
var - nome da variável
val - valor que a variável assume
```

2.2.11 short

`short` é um tipo de dados de 16 *bits*. Em todos os Arduinos `short` guarda valores de 16 *bits*. Possui o intervalo de valores de -32.768 a 32.767 (valor mínimo de -2^{15} e máximo de $(2^{15})-1$).

Exemplo

```
short ledPin = 13;
```

Sintaxe

```
short var = val;
```

```
onde:
var - nome da variável
val - valor que a variável assume
```

2.2.12 float

Basicamente são números com vírgula. Números de ponto flutuante são normalmente usados para aproximar números analógicos e contínuos por conta de sua maior precisão do que os números inteiros. Números de ponto flutuante podem ser tão qgrandes quanto $3.4028235E+38$ e tão pequeno quanto $-3.4028235E+38$. É possível guardar 32 *bits* (4 *bytes*) de informação. *Floats* possuem apenas de 6 a 7 dígitos de precisão. Diferentemente de outras plataformas, onde pode-se obter mais precisão usando `double` (em torno de 15 dígitos), no Arduino, `double` é do mesmo tamanho que `float`.
Números `float` não são exatos, e podem ter resultados estranhos quando comparados. Por exemplo, $6.0/3.0$ não é igual a 2.0. Operações aritméticas de ponto flutuante são muito mais devagares do que inteiros. Programadores normalmente convertem números de ponto flutuante para inteiros para aumentar a velocidade de seus códigos. Para tratar como `float`, deve-se adicionar o ponto do decimal, caso contrário, será tratado como inteiro.

Exemplo

```
float minha_variavel;
float sensorCalibrado = 1.117;
```

Sintaxe

```
float var = val;
```

onde:
var - nome da variável
val - valor que a variável assume

Exemplo de código

```
int x;
int y;
float z;

x = 1;
y = x / 2;            // y agora contém 0, int não guarda frações
z = (float)x / 2.0;   // z possui .5 (deve-se usar 2.0, ao invés de 2)
```

2.2.13 double

"Precisão dupla" para números de ponto flutuante. Nos Arduinos UNO e outras placas da ATMEGA, ocupa 4 *bytes*. Isto é, a implementação do *double* é exatamente a mesma do `float`, sem ganhos de precisão. No Arduino Due, `double` possui 8 *bytes* (64 *bits*) de precisão.

2.2.14 string - char array

`Strings` podem ser representadas de duas formas. Usando a estrutura de dados chamada `String` ou usando um vetor de `char`. Para mais detalhes do método que usa a estrutura `String`, que fornece mais funcionalidade no custo de memória, siga a subsecção 2.2.15.

Formas de se declarar *strings*

Tabela 2.3: *Strings*: como declarar?

Código	Forma de declarar
`char Str1[15];`	Vetor sem inicialização
`char Str2[8] = {'a', 'r', 'd', 'u', 'i', 'n', 'o'};`	Vetor com espaço extra, sem inicialização
`char Str3[8] = {'a', 'r', 'd', 'u', 'i', 'n', 'o', '\0'};`	Expicitando o caracter nulo
`char Str4[ ] = "arduino";`	Inicializar com uma string constante entre aspas, o compilador a ajustar a sequência de constantes e um caractere nulo de terminação
`char Str5[8] = "arduino";`	Inicializar a matriz com um tamanho explícito e string constante
`char Str6[15] = "arduino";`	Inicializar o array, deixando espaço extra para uma sequência maior

Terminação nula (*null*)

Geralmente, *strings* terminam com um caractere nulo (0, no código ASCII). Isso permite que funções como a `Serial.print()` "saibam" quando termina a *string*. Por outro lado, as funções continuam lendo os *bytes* de memória que na verdade não fazem parte da *string*. Isso significa que a *string* precisa ter um espaço para mais de um caracter do que o texto que se deseja escrever. É por isso que Str2 e Str5 precisam possuir oito caracteres, apesar de"arduino" possuir apenas sete - a última posição é automaticamente preenchida com um caracter nulo. Str4 será automaticamente ajustada para o tamanho de oito caracteres, um para o nulo. Em Str3, foi explicitamente incluído o caracter nulo (`\0`).

É possível ter *strings* sem um caracter final nulo (caso se no Str2 o tamanho discriminado fosse sete ao invés de oito). Isso deve ser evitado, pois algumas funções podem não funcionar. Muitos erros de ordem confusa de carcteres de vetores ocorrem por esse fato.

Aspas duplas ou simples?

Strings são sempre definidas com aspas duplas (`"Abc"`) e caracteres são sempre definidos com aspas simples (`'A'`).

Concatenando *strings* longas

```
char String[] = "primeira linha"
" segunda linha"
" etcetera";
```

Vetores de *strings*

É normalmente conveniente, quando se trabalha com grandes quantidades de texto, assim como um projeto de *display* LCD, usar um vetor de *strings*. *Strings* são por si só vetores, um exemplo de um vetor bidimensional.

No exemplo abaixo, o asterisco depois do `char` "`char*`" indica que é um vetor de ponteiros. Todo nome de vetor na verdade é um ponteiro, então, é preciso disso para fazer um vetor de vetores. Não é necessário entender de ponteiros para este exemplo.

Exemplo

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

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

void loop(){
for (int i = 0; i < 6; i++){
   Serial.println(Strings[i]);
   delay(500);
   }
}
```

2.2.15 String - object

A classe *String* permite que se manipule *strings* do texto de formas mais complexas do que com vetores. É possível concatenar *Strings*, anexá-los, procurar e substituir substrings e muito mais. É preciso mais memória do que uma matriz de caracteres simples, mas também é mais útil. Para referência, arrays de caracteres são chamados de *strings* com "s" minúsculo (s), e as instâncias da classe *String* são chamados de *strings* de "s" maiúsculo(S). Note que *strings* de constantes, indicadas com aspas duplas são tratadas como matrizes de caracteres, e não instâncias da classe *String*.

2.2.16 array

Um `array` (vetor) é uma coleção de variáveis que são acessadas por um número índice. Declarando um vetor (todos os métodos são válidos):

```
int inteiros[6];
int pinos[] = {2, 4, 8, 3, 6};
int valores[6] = {2, 4, -8, 3, 2};
char mensagem[3] = "ola";
```

É possivel declarar um vetor sem inicializá-lo, como em "inteiros".

Já em "pinos" o vetor foi declarado sem explicitar sem tamanho. O compilador conta os elementos e aloca um espaço de memória com o tamnho apropriado. Em "valores", há todas as informações, o tamanho e os valores do vetor. Para preencher um vetor com o tipo `char`, deve-se discriminar o tamanho e o conteúdo. Pode-se considerar cada espaço do vetor como uma "casa". Desta forma, um vetor com 6 elementos, possui como índices de 0 a 5. A primeira casa sempre possui índice como zero.

Atribuindo valores a um vetor

```
vet[0] = 10;
```

Recebendo um valor de um vetor

```
x = vet[4];
```

Vetores e `for`

Vetores normalmente são manipulados dentro de *loops* onde o contador do laço é usado como índice para cada elemento do vetor. Por exemplo, escrever os elementos do vetor na porta serial:

```
int i;
for (i = 0; i < 5; i = i + 1) {
  Serial.println(pinos[i]);
}
```

2.3 Conversão

2.3.1 char()

Converte um valor para o tipo de dados `char`.

Sintaxe

```
char(x)
```

Parâmetros

```
x: valor de qualquer tipo
```

Retorna

```
char
```

2.3.2 byte()

Converte um valor para o tipo de dados `byte`.

Sintaxe

```
byte(x)
```

Parâmetros

```
x: valor de qualquer tipo
```

Retorna

```
byte
```

2.3.3 int()

Converte um valor para o tipo de dados `int`.

Sintaxe

```
int(x)
```

Parâmetros

```
x: valor de qualquer tipo
```

Retorna

```
int
```

2.3.4 word()

Converte um valor para tipo de dados `word` ou cria uma paralvra de dois *bytes*.

Sintaxe

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

Parâmetros

```
x: valor de qualquer tipo
h: o byte mais à esquerda da palavra
l: o byte mais à direita da palavra
```

Retorna

```
word
```

2.3.5 long()

Converte um valor para o tipo de dados `long`.

Sintaxe

```
long(x)
```

Parâmetros

```
x: valor de qualquer tipo
```

Retorna

```
long
```

2.3.6 float()

Converte um valor para o tipo de dados `float`.

Sintaxe

```
float(x)
```

Parâmetros

```
x: valor de qualquer tipo
```

Retorna

```
float
```

2.4 Escopo de variáveis

2.4.1 Escopo de uma variável

Variáveis em C, usada no Arduino, ssui uma propriedade chamada de escopo. Essa é uma diferença a linguagens mais antigas como BASIC onde todas as variáveis são globais.
Variável globar é uma que pode ser visualizada(utilizada) por qualquer função em um programa. Variáveis locais são visíveis apenas para a função que foram declaradas. No ambiente do Arduino, qualquer variável declarada fora de uma função, é global. Quando programas começam a se tornar maiores e mais complexos, variáveis locais são a melhor solução para que cada função tenha acesso a suas próprias variáveis. Isso previne possíveis erros de programação onde uma função modifica as variáveis de outra função.
Algumas vezes é útil declarar e inicializar a variável dentro de um laço do tipo `for`. Desta forma, essa variável poderá ser acessada apenas dentro do laço.

Exemplo

```
int gPWMval;  // nenhuma função usará essa variável

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

```
void loop(){
  int i;    // "i" será visível apenas na função loop
  float f;  // "f" será visível apenas na função loop
  // ...

  for (int j = 0; j <100; j++){
  // a variável j pode apenas ser acessada dentro do laço for
  }
}
```

2.4.2 Estáticas

A palavra reservada `static` é usada para criar variáveis que são visíveis para apenas uma função. Variáveis locais são criadas e destruídas todas as vezes que uma função é chamada, variáveis estáticas (`static`) persistem mesmo após a chamada terminar, guardando a informação.
Variáveis declaradas como `static` são criadas e inicializadas apenas na primeira vez que a função é chamada.

Exemplo

```
int var1;
int var2;

void setup(){
// ...
}

void loop(){
// ...
}

int funcao(){
  static int var3;
   ações
}
```

2.4.3 Volátil

`volatile` é a palavra reservada conhecida como qualificadora, usualmente usada antes do tipo da variável, possui o intuito de modificar como o compilador e o programa irão tratar a variável. Declarar uma variável com `volatile` é uma diretiva para o compilador. O compilador é um *software* que traduz o código em C/C++, que envia as reais instruções para o chip Atmega do Arduino. Especificamente, ele direciona o compialor a carregar a variável da RAM e não do que está armazenado no registrador[5]. Dentro de certas condições, o valor da variável armazenada nos registradores pode ser impreciso. Uma variável pode ser declarada como `volatile` quando o valor possa ser trocado por algo presente no trecho de código, assim como por ações concomitantes.

Exemplo

```
// alterna o estado(acende ou apaga) do LED quando o pino de interrupção muda de estado
int pino = 13;
volatile int estado = LOW;

void setup(){
  pinMode(pino, OUTPUT);
  attachInterrupt(0, blink, CHANGE);
}

void loop(){
  digitalWrite(pino, estado);
}

void blink(){
  estado = !estado;
}
```

2.4.4 Constante

A palavra reservada `const` significa constante. É um qualificador variável que modifica o comportamento da variável, tornando uma variável "*read-only*". Isto significa que a variável pode ser utilizada, assim como qualquer outra variável do seu tipo, mas o seu valor não pode ser alterado. O compilador acusará erro no caso de uma tentativa de atribuir um valor a uma variável `const`.
Constantes definidas com a palavra reservada `const` obedecem as regras de escopo de variáveis que governam outras variáveis. Isso, e as armadilhas do uso de `#define`, faz com que a palavra reservada `const` um método superior para a definição de constantes. Assim, é preferível a usar `#define`.

[5] memória temporária onde as variáveis do programa são guardas e manipuladas.

Exemplo

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

// ....

x = pi * 2;    // pode-se usar constantes em operações aritméticas

pi = 7;        // errado! não se pode atribuir valores a contantes!
```

A tabela 2.4 demonstra quando usar `#define` **ou** `const`:

Tabela 2.4: define e const: quando usar?

tipo de variável	`#define`	`const`
Contantes numéricas	√	√
`Strings`	√	√
Vetores		√

2.5 Utilitários

2.5.1 sizeof()

O operador `sizeof()` retorna o número de *bytes* de um tipo de variável ou o número de *bytes* ocupados em um vetor.

Sintaxe

```
sizeof(variable)
```

Parâmetros

```
variável: qualquer tipo de variável ou vetor (exemplo: int, float, byte)
```

Exemplo

O operador `sizeof` normalmente é usada para lidar com vetores (assim como *strings*), onde é conveniente mudar o tamanho de um vetor sem quebrar outras partes do programa. O programa abaixo imprime um texto de um caracter por vez.

```
char minhaString[] = "teste";
int i;

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

void loop() {
  for (i = 0; i < sizeof(minhaString) - 1; i++){
    Serial.print(i, DEC);
    Serial.print(" = ");
    Serial.write(minhaString[i]);
    Serial.println();
  }
  delay(5000); // reduz a velociade do programa
}
```

Note que `sizeof` retorna o número total de *bytes*. Então se for um tipo de variável como `int`, deve-se usar um *loop* do tipo `for`. Repare que uma `string` bem formada termina com valor de NULL, que no código ASCII equivale a 0 (zero).

```
for (i = 0; i < (sizeof(meusInteiros)/sizeof(int)) - 1; i++) {
  // faça algo com meusInteiros[i]
}
```

Capítulo 3

Funções

3.1 Entrada e saída digital

3.1.1 pinMode()

Configura um pino específico para definir o comportamento entre *input* ou *output*. No Arduino 1.0.1, é possivel habilitar os resistores internos *pullup* no modo `INPUT_PULLUP`. O modo *INPUT* explicitamente desabilita os *pullups* internos.

Sintaxe

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

Parâmetros

```
pino: o número do pino escolhido a ser usado
modo: INPUT, OUTPUT, ou INPUT_PULLUP
```

Exemplo

```
int ledPin = 13;                 // LED conectado no pino 13

void setup(){
  pinMode(ledPin, OUTPUT);      // definindo o pino como output
}

void loop(){
  digitalWrite(ledPin, HIGH);   // liga o LED
  delay(1000);                  // aguarda um segundo
  digitalWrite(ledPin, LOW);    // desliga o LED
  delay(1000);                  // aguarda um segundo
}
```

3.1.2 digitalWrite()

Escreve o valor de `HIGH` ou `LOW` em um pino digital. Se o pino for configurado como `OUTPUT` com `pinMode()`, essa tensão será aplicada a seu valor correspondente: 5V (ou 3.3V nas placas de 3.3V) para `HIGH`, 0V (ground) para `LOW`. O pino é configurado como `INPUT`, `digitalWrite()` irá habilitar (`HIGH`) ou desabilitar (`LOW`) o resistor interno *pullup* no pino. Recomenda-se configurar o `pinMode()` como `INPUT_PULLUP` para permitir que a resistência interno *pull-up*. Ao configurar `pinMode()` como `OUTPUT`, e conectar o LED a um pino, ao chamar `digitalWrite(HIGH)`, o LED pode acender de forma fraca. Sem explicitar o modo do pino, `digitalWrite()` irá habilitar os resistores internos *pull-up*, que age como um resistor largo limitador de corrente.

Sintaxe

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

Parâmetros

```
pino: número do pino
valor: HIGH ou LOW
```

Exemplo

```
int ledPin = 13;                 // LED conectado ao pino digital  13

void setup(){
  pinMode(ledPin, OUTPUT);      // define o pino digital como output
}

void loop(){
  digitalWrite(ledPin, HIGH);   // liga o LED
  delay(1000);                  // aguardar um segundo
  digitalWrite(ledPin, LOW);    // desliga o LED
  delay(1000);                  // aguardar um segundo
}
```

3.1.3 digitalRead()

Faz a leitura de um pino digital específico, tanto `HIGH` ou `LOW`.

Sintaxe

```
digitalRead(pino)
```

Parâmetros

```
pino: o número do pino digital que se deseja fazer a leitura (int)
```

Retorna

`HIGH` ou `LOW`

Exemplo

```
Pino 13 possui o mesmo valor do pino 7, declarado como input.

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

void setup(){
  pinMode(ledPin, OUTPUT);      // define o pino digital 13 como output
  pinMode(inPin, INPUT);      // define o pino digital 7 como input
}

void loop(){
  val = digitalRead(inPin);   // faz a leitura do pino definido como input
  digitalWrite(ledPin, val);
}
```

Dica de código

Se o pino não estiver conectado a nada, `digitalRead()` pode retornar `HIGH` ou `LOW` (isso pode ser trocando de forma randômica).

3.2 Entrada e saída analógica

3.2.1 analogReference()

Configura a tensão de referência usada na entrada analógica (isto é, o valor usado como topo do intervalo). As opções possíveis estão dispostas na Tabela 3.2.1.

Tabela 3.1: Opções da função **analogReference()**.

Opções	Ação executada
`DEFAULT`	a referência padrão é de 5 volts (nas placas de Arduino de 5V) ou 3.3 volts (nas placas de Arduino de 3.3V)
`INTERNAL`	igual a 1,1 volts no ATmega168 ou ATmega328 e 2,56 volts no ATmega8 (não disponível no Arduino Mega)
`INTERNAL1V1`	usa 1.1V como referência (apenas Arduino Mega)
`INTERNAL2V56`	usa 2.56V como referência (apenas Arduino Mega)
`EXTERNAL`	a tensão aplicada ao pino AREF (de 0 a 5V apenas) é usada como referência.

Parâmetros

```
type: qual tipo de referência que será usada
(DEFAULT, INTERNAL, INTERNAL1V1, INTERNAL2V56, ou EXTERNAL).
```

Dica de código

Depois de trocar a referência analóica, as primeiras leituras de `analogRead()`(3.2.2) podem não ser precisas.

Não se deve usar menos do que 0V ou mais do que 5V para referências externas nos pinos usados com AREF! Caso se use uma referência externa com o pino,deve ser definida a referência analógica como `EXTERNAL` antes de se fazer a leitura com `analogRead()`. Por outro lado, isso encurtará a referência de tensão ativa (gerada internamente) e o pino usado com o AREF, assim podendo danificar o microcontrolador na placa Arduino. Alternativamente, é possível ligar a tensão de referência externa para o pino AREF através de um resistor de 5K, que permite alternar entre tensões de referência externas e internas. Nota-se que a resistência alterará a tensão que é usada como a referência, porque existe uma resistência de 32K interna sobre o pino AREF. Os dois atuam como um divisor de tensão, de modo que, por exemplo, 2,5 V aplicada através da resistência irá produzir $2,5 * 32/(32 + 5) =\sim 2{,}2$ V, no pino AREF.

3.2.2 analogRead()

Faz a leitura de valores de um pino analógico. A placa possui 6 canais (8 canais no Mini e Nano, 16 no Mega) de 10 *bits* no conversor analógico para digital. Isso significa que as tensões de entrada serão mapeadas em valores entre 0 e 5 volts para valores inteiros de 0 a 1023. Isso gera uma resolução entre as leituras de: 5 volts por 1024 unidades, ou 0,0049 volts (4,9 mV) por unidade. O alcance e a resolução de entrada podem ser alterados usando `analogReference()`. Demora cerca de 100 microssegundos (0,0001s) para ler uma entrada analógica, então a taxa máxima de leitura é de cerca de 10.000 vezes por segundo.

Sintaxe

```
analogRead(pin)
```

Parâmetros

```
pin: o número de entradas analógicas a serem lidas (0 a 5 na maioria
das placas, 0 a 7 nos Arduinos Mini e Nano, 0 a 15 no Arduino Mega)
```

Retorna

```
int (0 a 1023)
```

Dica de código

Se o pino anlógico for declarado mas não estiver conectado a nada, o valor que `analogRead()` retornará será baseado em um número de fatores, como, por exemplo, os valores das outras entradas analógicas.

Exemplo

```
int analogPin = 3;
int val = 0;           // variável para guardar o valor lido

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

void loop(){
  val = analogRead(analogPin);    // pino de entrada de leitura
  Serial.println(val);
  }
```

3.2.3 analogReadResolution()

`analogReadResolution()` é uma extensão do pino analógico para o Arduino Due. Define o tamanho (em bits) do valor determinado por `analogRead()`. Seu padrão é 10 bits (determina valores entre 0-1023) para versões anteriores com placas baseadas em AVR. O Due tem 12-bit ADC de capacidade que pode ser acessada mudando a resolução para 12. Isso vai determinar valores da `analogRead()` entre 0 e 4095.

Sintaxe

```
analogReadResolution(bits)
```

Parâmetros

```
bits: determina a resolução (em bits) do valor determinado pela
função analogRead(). É possível determinar isso de 1 à 32. Pode-se
definir resoluções mais altas que 12, mas os valores determinados
por analogRead() vão sofrer aproximações. Veja a nota abaixo para
detalhes.
```

Nota

Se o valor de `analogReadResolution()` determinado for um valor maior do que a capacidade da placa, o Arduino apenas vai informar, na sua maior resolução, os bits extras com zeros. Por exemplo: ao usar o Due com `analogReadResolution(16)` se obtém uma aproximação de um número de 16-bit com os primeiros 12 bits contendo a leitura real ADC e os 4 últimos contendo zeros. Se o valor de `analogReadResolution()` determinado for um valor menor do que a capacidade da placa, os valores extras de bits significantes na leitura ADC serão descartados.

Dica de código

Usar uma resolução de 16-bit (ou qualquer outra maior que a capacidade do hardware em questão) permite escrever programas que automaticamente manipulam dispositivos com uma resolução ADC maior, quando estes estiverem disponíveis em placas futuras, sem mudar a linha do código.

Exemplo

```
void setup() {
 Serial.begin(9600);   // abre a conexão serial
}

void loop() {
 analogReadResolution(10);
 // lê a entrada em A0 na resolução padrão
 //(10 bits) e envia a conexão serial
 Serial.print("ADC 10-bit (default) : ");
 Serial.print(analogRead(A0));

 analogReadResolution(12);  // muda a resulação para 12 bits e lê A0
 Serial.print(", 12-bit : ");
 Serial.print(analogRead(A0));

 analogReadResolution(16);  // muda a resulação para 16 bits e lê A0
 Serial.print(", 16-bit : ");
 Serial.print(analogRead(A0));

 analogReadResolution(8);  // muda a resulação para 8 bits e lê A0
 Serial.print(", 8-bit : ");
 Serial.println(analogRead(A0));

 delay(100);  // uma pausa para não lotar a serial monitor
}
```

3.2.4 analogWrite() - *PWM*

Escreve um valor analógico (onda PWM) para o pino. Pode ser usada para acender um LED variando sua intensidade luminosa ou acionar um motor em diferentes velocidades. Depois de chamar `analogWrite()`, o pino vai gerar uma onda quadrada constante do ciclo de trabalho especificado até a próxima chamada `analogWrite()` (ou uma chamada `digitalRead()` ou `digitalWrite()` no mesmo pino). A frequência do sinal PWM na maioria dos pinos é aproxidamente 490 Hz. No Uno e placas similares, os pinos 5 e 6 têm uma frequência de aproximadamente 980 Hz. Na maioria dos Arduinos (com ATmega168 ou ATmega328), a função funciona nos pinos 3, 5, 6, 9, 10, and 11. No Arduino Mega, funciona nos pinos 2 - 13 e 44 - 46. Em Arduinos mais antigos com ATmega8 só funciona `analogWrite()` nos pinos 9, 10, e 11.

No Arduino Due `analogWrite()` funciona nos pinos 2 até 13, somados aos pinos DAC0 e DAC1. Ao contrário dos pinos PWM, DAC0 e DAC1 são conversores Digital para Analógico, e funcionam como verdadeiros pinos de saída analógicos. Não é necessário chamar `pinMode()` para determinar o pino como de saída antes de chamar `analogWrite()`.
A função `analogWrite()` não tem relação com os pinos analógicos ou com a função `analogRead()`.

Sintaxe

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

Parâmetros

```
pino: o número do pino escolhido a ser usado.
valor: o ciclo de trabalho: entre 0 (sempre desligado) e 255.
```

Nota

As saídas PWM, geralmente nos pinos 5 e 6, vão ter ciclos de trabalho maiores que o esperado. Isso por conta das interações com as funções `millis()` e `delay()`, que compartilham do mesmo tempo interno usado para gerar essas saídas PWM. Isso pode ser observado principalmente em definições de ciclos de trabalho baixos (ex.: 0 - 10) que podem resultar em valores de 0 sem desligar totalmente as saídas nos pinos 5 e 6.

Exemplo

```
Determinar a saída do LED proporcionalmente ao valor lido no potenciômetro.


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

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

void loop()
{
 val = analogRead(analogPin);   // lê o pino de entrada
 analogWrite(ledPin, val / 4);  // analogRead valores vão de 0 à 1023,
                                //analogWrite vão de 0 à 255
}
```

3.2.5 analogWriteResolution()

`analogWriteResolution()` é a extensão da Analog API para o Arduino Due.
`analogWriteResolution()` determina a resolução da função `analogWrite()`. O padrão será

de 8 bits (valores ente 0-255) para versões com compatibilidade com placas baseadas em AVR. O Due tem as seguintes capacidades de hardware:
12 pinos que terão padrão de PWM 8-bit, como as placas baseadas em AVR. Isto pode ser trocado para uma resolução de 12-bit.
2 pinos com 12-bit DAC (Conversor Digital-Analógico).
Ao definir a resolução para 12, pode-se usar `analogWrite()` com valores entre 0 e 4095 para analisar toda a resolução DAC ou determinar o sinal PWM.

Sintaxe

```
analogWriteResolution(bits)
```

Parâmetros

```
bits: determina a resolução em (em bits) do valor usado na função
analogWrite(). O valor pode variar de 1 à 32. Se a resolução
escolhida for maior ou menor que a capacidade do hardwar da placa,
o valor usado na analogWrite() vai ser cortado se for muito alto
ou substituído por zeros se for muito baixo. Veja a nota abaixo
para mais detalhes.
```

Nota

Se determinou-se o valor `analogWriteResolution()` para um valor mais alto que a capacidade da placa, o Arduino vai discartar os bits extras. Por exemplo: usando o Due com `analogWriteResolution(16)` em um pino 12-bit DAC, somente os primeiros 12 bits dos valores passados para `analogWrite()` vão ser usados e os últimos 4 bits vão ser descartados. Se determinou-se o valor `analogWriteResolution()` para um valor mais baixo que a capacidade da placa, os bits que faltam vão ser substituídos por zeros para preencher o tamanho requirido pelo hardware. Por exemplo: ao usar o Due com `analogWriteResolution(8)` em um pino 12-bit DAC, o Arduino vai adicionar 4 zero bits para o valor de 8-bit usado na analogWrite() para obter os 12 bits requeridos.

Exemplo

```
void setup(){
 // abre a conexão serial
 Serial.begin(9600);
 // determina os pinos digitais como de saída
 pinMode(11, OUTPUT);
 pinMode(12, OUTPUT);
 pinMode(13, OUTPUT);
}

void loop(){
 // lê a entrada em A0 e descreve o pino PWM
 // com um LED ligado
 int sensorVal = analogRead(A0);
 Serial.print("Analog Read) : ");
 Serial.print(sensorVal);

 // a resolução padrão PWM
 analogWriteResolution(8);
 analogWrite(11, map(sensorVal, 0, 1023, 0 ,255));
 Serial.print(" , 8-bit PWM value : ");
 Serial.print(map(sensorVal, 0, 1023, 0 ,255));

 // muda a resolução PWM para 12 bits
 // A resolução de 12 bits completa só é suportada no Duemilanove
 analogWriteResolution(12);
 analogWrite(12, map(sensorVal, 0, 1023, 0, 4095));
 Serial.print(" , 12-bit PWM value : ");
 Serial.print(map(sensorVal, 0, 1023, 0, 4095));

 // muda a resolução PWM para 4 bits
 analogWriteResolution(4);
 analogWrite(13, map(sensorVal, 0, 1023, 0, 127));
 Serial.print(", 4-bit PWM value : ");
 Serial.println(map(sensorVal, 0, 1023, 0, 127));
```

```
 delay(5);
}
```

3.3 Entrada e saída avançada

3.3.1 Tone()

Gera uma onda quadrada de frequência específica (e 50% do ciclo de trabalho) em um pino. A duração pode ser específica, por outro lado a curva continua até chamar `noTone()`. O pino pode ser conectado à um piezo buzzer ou à outros semelhantes para reproduzir sons. Somente um som pode ser gerado em um tempo. Se o som já está tocando em um pino diferente, chamar `tone()` não terá efeito. Se o som está tocando no mesmo pino, chamar `tone()` vai mudar a frequência.
Usar a função `tone()` vai interferir com a saída PWM nos pinos 3 e 11 (em placas que não sejam a Mega). Não é possível gerar sons mais baixos que 31 Hz.

Nota

Se o objetivo for tocar diferentes tons em diversos pinos, é necessário chamar `noTone()` em um pino antes de chamar `tone()` no próximo pino.

Sintaxe

```
tone(pino, frequência)
tone(pino, frequência, duração)
```

Parâmetros

```
pino: o pino que gera o som.
frequência: a frequência do som em hertz.
duração: a duração do som em milisegundos (opcional).
```

3.3.2 noTone()

Para a geração de uma onda quadrada gerada por `tone()`. Não tem efeito se nenhum som está sendo gerado.

Nota

Se o objetivo for tocar diferentes tons em diversos pinos, é necessário chamar `noTone()` em um pino antes de chamar `tone()` no próximo pino.

Sintaxe

```
noTone(pino)
```

Parâmetros

```
pino: o pino que vai parar de gerar som.
```

3.3.3 shiftOut()

Desloca um bit de cada vez de um byte de dados . Começa do maior (ex.: mais a esquerda) ou menor (ex.: mais a direita) bit significativo. Cada bit é escrito por turno a um pino de dados, após o pino(clockPin) receber um pulso (primeiro HIGH, depois LOW) para indicar que o bit é válido. Se há uma interação com um dispositivo que é ativado pelo aumento das extremidades, é necessário ter certeza que o pino está desativado antes de chamar `shiftOut()`, ex.: com a chamada `digitalWrite(clockPin, LOW)`.

Nota

Isso é uma implementação de *software*; veja também a biblioteca SPI, que providencia uma implementação de *hardware* que é mais rápida mas funciona somente em pinos específicos.

Sintaxe

```
shiftOut(dataPin, clockPin, bitOrder, value)
```

Parâmetro

```
dataPin: o pino em que haverá a saída de cada bit (int).
clockPin: o pino de alternância uma vez que o dataPin
foi definido para o valor correto (int).
bitOrder: que vai deslocar os bits; tanto MSBFIRST ou LSBFIRST
(Primeiro o bit mais signicativo, ou, primeiro o bit menos signicativo).
value: os dados para deslocar. (byte).
```

Dica de código

O dataPin e clockPin devem ser registrados como de saída ao chamar `pinMode()`. `shiftOut` é escrita na saída 1 byte (8 bits), então requer uma operação de duas etapas para valores maiores que 255.

```
// Faça isso para MSBFIRST serial
int data = 500;
// desloque highbyte
shiftOut(dataPin, clock, MSBFIRST, (data >> 8));
// desloque lowbyte
shiftOut(dataPin, clock, MSBFIRST, data);

// Ou faça isso para LSBFIRST serial
data = 500;
// desloque lowbyte
shiftOut(dataPin, clock, LSBFIRST, data);
// desloque highbyte
shiftOut(dataPin, clock, LSBFIRST, (data >> 8));
```

Exemplo

```
Para o circuito, veja o tutorial sobre como controlar um registro de troca 74HC595.
//**************************************************************//
//  Nome    : shiftOutCode, Hello World                          //
//  Autor   : Carlyn Maw,Tom Igoe                               //
//  Data    : 25 Oct, 2006                                       //
//  Versão  : 1.0                                               //
//  Notas   : Código para usar um registro de troca 74HC595      //
//          : para ter valores de 0 à 255                        //
//**************************************************************

//Pino conectado ao ST_CP do 74HC595
int latchPin = 8;
//Pino conectado ao SH_CP do 74HC595
int clockPin = 12;
//Pino conectado ao DS do 74HC595
int dataPin = 11;
```

```
void setup() {
 //determinar os pinos como de saída
 pinMode(latchPin, OUTPUT);
 pinMode(clockPin, OUTPUT);
 pinMode(dataPin, OUTPUT);
}

void loop() {
 //sequência de contagem
 for (int j = 0; j < 256; j++) {
   //deslique o latchPin e mantenha assim enquanto
   //estiver havendo transmissão
   digitalWrite(latchPin, LOW);
   shiftOut(dataPin, clockPin, LSBFIRST, j);
   //ligue o latchPin para sinilizar ao chip que ele
   //não precisa mais receber informações
   digitalWrite(latchPin, HIGH);
   delay(1000);
 }
}
```

3.3.4 shiftIn

Muda um bit de cada vez de um byte de dados. Começa do maior (ex.: mais a esquerda) ou menor(ex.: mais a direita) bit significativo. Para cada bit, o pino(clockPin) é ativado (HIGH), o próximo bit é lido da linha de dados, e então o pino é desativado(LOW). Se há uma interação com um dispositivo que é ativado pelo aumento das extremidades, é necessário ter certeza que o pino está desativado antes de chamar `shiftOut()`, ex.: com a chamada `digitalWrite(clockPin, LOW)`.

Nota

Isso é uma implementação de *software*; o Arduino também oferece a biblioteca SPI, que usa uma implementação de *hardware* que é mais rápida mas funciona somente em pinos específicos.

Sintaxe

```
shifIn(dataPin, clockPin, bitOrder)
```

Parâmetro

```
dataPin: o pino em que haverá a entrada de cada bit (int).
clockPin: o pino de alternância do sinal a ser lido do dataPin (int).
bitOrder: que vai deslocar os bits; tanto MSBFIRST ou LSBFIRST
(Primeiro o bit mais signicativo, ou, primeiro o bit menos signicativo).
```

Retorna

O valor lido (byte).

3.3.5 pulseIn

Lê um pulso (tanto `HIGH` ou `LOW`) de um pino. Por exemplo, se o valor é `HIGH`, `pulseIn()` espera pelo pino para ligar, inicia a contagem, então espera o pino para desligar e parar a contagem. Retorna a duração do pulso em microsegundos. Desiste e retorna 0 se nenhum pulso começa dentro de um tempo limite especificado. A contagem dessa função foi determinada empíricamente e provavelmente vai apresentar erros em longos pulsos. Funciona em pulsos de 10 microsegundos à 3 minutos de duração.

Sintaxe

```
pulseIn(pino, valor)
pulseIn(pino, valor, timeout)
```

Parâmetro

```
pino: o número do pino em que lê-se o pulso. (int)
valor: tipo do pulso a ser lido: tanto HIGH ou LOW. (int)
timeout (opcional): o número de microsegundos para esperar
o pulso começar; o padrão é um segundo. (unsigned long)
```

Retorna

A duração do pulso (em microsegundos) ou 0 se nenhum pulso começar antes do tempo limite (`unsigned long`).

Exemplo

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

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

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

3.4 Temporização

3.4.1 millis()

Retorna o número em milisegundos desde que o Arduino começou a rodar o programa em questão. Este número vai sobrecarregar (voltar para zero), aproximadamente após 50 dias.

Retorna

Número em milisegundos desde que o programa começou (`unsigned long`).

Exemplo

```
unsigned long time;

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

void loop(){
 Serial.print("Time: ");
 time = millis();
 //printa o tempo desde que o programa começou
 Serial.println(time);
 //espera um segundo para não enviar grande quantidade de dados
 delay(1000);
}
```

Dica de código:

O paramêtro para `millis` é um `unsigned long`, erros podem ser gerados se o programa usar outros tipos de dados como `int`.

3.4.2 micros()

Retorna o número em microsegundos desde que o Arduino começou a rodar o programa em questão. O número vai sobrecarregar (voltar a zero), aproximadamente após 70 minutos. Em 16 MHz placas de Arduino (ex.: Duemilanove e Nano), essa função tem a resolução de quatro microsegundos (o valor retornado é sempre um múltiplo de quatro). Em 8 MHz placas de Arduino (ex.: o LilyPad), esta função tem uma resolução de oito microsegundos.

Nota

Há 1.000 microsegundos em um milisegundo e 1.000.000 microsegundos em um segundo.

Retorna

Número de microsegundos desde que o programa começou (`unsigned long`).

Exemplo

```
unsigned long time;

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

void loop(){
 Serial.print("Time: ");
 time = micros();
 //printa o tempo desde que o programa começou
 Serial.println(time);
 //espera um segundo para não enviar grande quantidade de dados
 delay(1000);
}
```

3.4.3 delay()

Pausa o programa na quantidade de tempo (em milisegundos) especificada como parâmetro. (Há 1000 milisegundos em um segundo).

Sintaxe

```
delay(ms)
```

Parâmetro

```
ms: o número em milisegundos da pausa (unsigned long)
```

Exemplo

```
int ledPin = 13;                  // LED conectado ao pino digital 13

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

void loop()
{
 digitalWrite(ledPin, HIGH);   // acende o LED
 delay(1000);                  // espera um segundo
 digitalWrite(ledPin, LOW);    // apaga o LED
 delay(1000);                  // espera um segundo
}
```

Nota

Enquanto é fácil fazer um LED piscar com a função `delay()`, e muitos programas usam pequenos delays para tarefas como *debouncing*, o uso do `delay()` em um código tem desvantagens significativas. Leitura de sensores, cálculos matemáticos, ou manipulações de pinos podem continuar durante a função `delay`, então, com efeito, isso traz mais outras atividades para fazer uma pausa. Para abordagens alternativas de controlar o tempo, existe, por exemplo, a função `millis()`. Programadores mais experiente procuram evitar o uso do `delay()` para a temporização de eventos com mais de 10 milisegundos, menos no código do Arduino que é muito simples.
No entanto, certas coisas continuam enquanto a função `delay()` controla o Atmega chip, porque a função delay+ não desabilita interrupções. A comunicação Serial que aparece no pino RX é gravada, valores PWM (`analogWrite`) e estados de pinos são mantidos, e interrupções vão trabalhar como deveriam.

3.4.4 delayMicroseconds()

Pausa o programa na quantidade de tempo (em microsegundos) especificada como parâmetro. Há mil microsegundos em um milisegundo, e um milhão de microsegundos em um segundo. Atualmente, o maior valor que vai produzir uma pausa exata é 16383. Isso pode mudar em futuros lançamentos de Arduino. Para pausas maiores que alguns milhares de segundos, deve-se usar `delay()`.

Sintaxe

```
delayMicroseconds(us)
```

Parêmetro

```
us: o número em microsegundo da pausa (unsigned int)
```

Exemplo

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

void setup()
{
 pinMode(outPin, OUTPUT);      // determina o pino digital como de saída
}

void loop()
{
 digitalWrite(outPin, HIGH);   // ativa o pino
 delayMicroseconds(50);        // pausa de 50 microsegundos
 digitalWrite(outPin, LOW);    // desativa o pino
 delayMicroseconds(50);        // pausa de 50 microsegundos
}

Configura o pino 8 para trabalhar como um pino de saida.
Manda pulsos com um período de 100 microsegundos.
```

Nota

Essa função trabalha com muita precisão na faixa de 3 microsegundos para cima. Não é possível assegurar que `delayMicroseconds` vai funcionar precisamente para pausas menores de tempo. A partir do Arduino 0018, `delayMicroseconds()` não desativa interrupções.

3.5 Funções matemáticas

3.5.1 min()

Calcula o mínimo de dois números.

Parâmetro

`x`: o primeiro número, qualquer tipo de dado
`y`: o segundo número, qualquer tipo de dado

Retorno

O menor de dois números.

Exemplos

```
sensVal = min(sensVal, 100);
// atribui sensVal ao menor entre sensVal ou 100
// assegurando que ele nunca fica acima de 100.
```

Nota

`max()` é muito usada para restringir o menor termo do intervalo de uma variável, enquanto `min()` é usada para restringir o maior termo do intervalo.

Dica de código

Por causa do modo que a função `min()` é executada, é necessário evitar usar outras funções dentro dos colchetes, isso pode levar a resultados incorretos.

```
min(a++, 100);   // evitar isto - produz resultados incorretos

a++;
min(a, 100);    // usar isto - manter outras operações fora da função
```

3.5.2 max()

Calcula o máximo entre dois números.

Parâmetro

`x: o primeiro número, qualquer tipo de dado`
`y: o segundo número, qualquer tipo de dado`

Retorna

O maior de dois valores do parâmetro.

Exemplos

```
sensVal = max(senVal, 20);
// atribui sensVal ao maior entre sensVal ou 20
// assegurando que ele nunca fique abaixo de 20
```

Nota

`max()` é muito usada para restringir o menor termo do intervalo de uma variável, enquanto `min()` é usada para restringir o maior termo do intervalo.

Dica de código

Por causa do modo que a função `max()` é executada, é necessário evitar usar outras funções dentro dos colchetes, isso pode levar a resultados incorretos.

```
max(a--, 100);   // evitar isto - produz resultados incorretos

a--;
max(a, 100);    // usar isto - manter outras operações fora da função
```

3.5.3 abs()

Computa o valor absoluto de um número.

Parâmetro

`x: o número`

Retorna

x: Se x for maior ou igual a 0.
-x: Se x for menor que 0.

Dica de código

Por causa do modo que a função `abs()` é executada, é necessário evitar usar outras funções dentro dos colchetes, isso pode levar a resultados incorretos.

```
abs(a++);   // evitar isto - produz resultados incorretos

a++;
abs(a);    // usar isto - manter outras operações fora da função
```

3.5.4 constrain()

Restringe um número a ficar dentro de um intervalo.

Parâmetro

```
x: o número a restringer, todo tipo de dado
a: o menor termo do intervalo, todo tipo de dado
b: o maior termo do intervalo, todo tipo de dado
```

Retorna

x: Se x está entre a e b
a: Se x é menor que a
b: Se x é maior que b

Exemplo

```
sensVal = constrain(sensVal, 10, 150);
// limites variam de valores de sensores entre 10 e 150
```

3.5.5 map()

Remapeia um número de um intervalo para outro. Isto é, um valor de *fromLow* ficaria mapeado para *toLow*, um valor *fromHigh* para *toHigh*, valores *in-between* para valores *in-between*, etc. Não restringe valores para dentro do intervalo, porque valores fora do intervalo são, por vezes, destinados e úteis. A função `constrain()` pode ser usada tanto antes ou depois desta função, se o termo do intervalo for desejado. Note que o "limite inferior"de qualquer intervalo pode ser maior ou menos que o "limite superior", então, a função `map()` pode ser usada para inverter um intervalo de números, por exemplo:
`y = map(x, 1, 50, 50, 1);`
A função também lida com números negativos, para isso, este exemplo:
`y = map(x, 1, 50, 50, -100);`
Isso também é válido e funciona bem. A função `map()` usa matemática inteira, então, não vai gerar frações. Restos fracionarios são cortados, e não são arredondados.

Parâmetro

```
value: o número a mapear
fromLow: o limite inferior do intervalo atual do valor.
fromHigh: o limite superior do intervalo atual do valor.
toLow: o limite inferior do intervalo desejado do valor.
toHigh: o limite superior do intervalo desejado do valor.
```

Retorna

O valor mapeado.

Exemplo

```
/* Mapeia um valor analógico para 8 bits (0 to 255) */
void setup() {}

void loop()
{
 int val = analogRead(0);
 val = map(val, 0, 1023, 0, 255);
 analogWrite(9, val);
}
```

Nota

Matematicamente, aqui está toda a função:

```
long map(long x, long in_min, long in_max, long out_min, long out_max)
{
 return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
}
```

3.5.6 pow()

Calcula o valor de um número elevado a um potência. `pow()` pode ser usada para elevar um número a uma potência fracionada. Isto e útil para gerar mapeamento exponencial para valores ou curvas.

Parâmetro

```
base: o número (float)
exponent: a potência que a base é elevada (float)
```

Retorna

O resultado de uma exponenciação (double).

3.5.7 sqrt()

Calcula a raiz quadrada de um número.

Parâmetro

```
x: o número, qualquer tipo de dado
```

Retorna

A raís quadrada do número (double).

3.5.8 sin()

Calcula o seno de em ângulo (em radianos). O resultado varia de -1 à 1.

Parâmetro

`rad: o ângulo em radianos (float)`

Retorna

O seno do ângulo (double).

3.5.9 cos()

Calcula o cosseno de em ângulo (em radianos). O resultado varia de -1 à 1.

Parâmetro

`rad: o ângulo em radianos (float)`

Retorna

O cosseno do ângulo (double).

3.5.10 tan()

Calcula a tangente de em ângulo (em radianos). O resultado varia de infinito negativo à infinito.

Parâmetro

`rad: o ângulo em radianos (float)`

Retorna

A tangente do ângulo (double).

3.5.11 randomSeed()

`randomSeed()` inicia o gerador de números pseudo-aleatório, fazendo com que comece em um ponto arbitrário na sua sequência aleatória. Esta sequência, embora muito longa e aleatória, é sempre a mesma. Se é importante para a sequência de valores gerados por `random()` diferir, em execuções subsequentes de um código, use `randomSeed()` para iniciar o gerador de números aleatórios com uma entrada bastante aleatória, assim como `analogRead()` em um pino desconectado. Por outro lado, isso pode, ocasionalmente, ser útil para usar sequências pseudo-aleatórias que repitam de maneira exata. Isso pode ser realizado chamando `randomSeed()` com um número fixo, antes de começar a sequência aleatória.

Parâmetro

`long, int - transmite um número para dar ínicio.`

Exemplo

```
long randNumber;

void setup(){
 Serial.begin(9600);
 randomSeed(analogRead(0));
}

void loop(){
 randNumber = random(300);
 Serial.println(randNumber);

 delay(50);
}
```

3.5.12 random()

A função `random()` gera números pseudo-aleatórios.

Sintaxe

```
random(max)
random(min, max)
```

Parâmetro

```
min - limite inferior do valor aleatório, inclusive (opcional)
max - limite superior do valor aleatório, exclusivo
```

Retorno

Um número aleatório entre min e max-1 (long).

Nota

Se é importante para a sequência de valores gerados por `random()` diferir, em execuções subsequentes de um código, use `randomSeed()` para iniciar o gerador de números aleatórios com uma entrada bastante aleatória, assim como `analogRead()` em um pino desconectado. Por outro lado, isso pode, ocasionalmente, ser útil para usar sequências pseudo-aleatórias que repitam de maneira exata. Isso pode ser realizado chamando `randomSeed()` com um número fixo, antes de começar a sequência aleatória.

Exemplo

```
long randNumber;

void setup(){
```

```
 Serial.begin(9600);

 // se o pino de entrada analógico 0 é desconectado,
// o retorno random analog vai causar a chamada da randomSeed()
// para gerar diferentes números iniciais cada vez que o código rodar.
 // randomSeed() vai embaralhar a função random.
 randomSeed(analogRead(0));
}

void loop() {
 // printa um número aleatório de 0 à 299.
 randNumber = random(300);
 Serial.println(randNumber);

 // printa um número aleatório de 10 à 19.
 randNumber = random(10, 20);
 Serial.println(randNumber);

 delay(50);
}
```

3.6 Bits e bytes

3.6.1 lowByte()

Extrai o byte de menor ordem (mais a direita) de uma variável (ex.: uma palavra).

Sintaxe

```
lowByte(x)
```

Parâmetro

```
x: um valor de qualquer tipo
```

Retorna

byte.

3.6.2 highByte()

Extrai o byte de maior ordem (mais a esquerda) de uma palavra (ou o segundo menor byte de um tipo de dado maior).

Sintaxe

```
highByte(x)
```

Parâmetro

```
x: um valor de qualquer tipo
```

Retorna

Byte.

3.6.3 bitRead()

Lê um bit de um número.

Sintaxe

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

Parâmetro

```
x: o número a partir do qual é feita a leitura.
n: o bit a ser lido, começando do 0 para o bit
menos significativo (mais a direita).
```

Retorna

O valor de um bit (0 ou 1).

3.6.4 bitWrite()

Escreve um bit de uma variável numérica.

Sintaxe

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

Parâmetro

```
x: a variável numérica a ser escrita.
n: o bit do número a ser escrito, começando do 0
para o bit menos significativo (mais a direita).
b: o valor a ser escrito para o bit (0 ou 1).
```

3.6.5 bitSet()

Determina (escreve um 1 para) um bit de uma variável numérica.

Sintaxe

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

Parâmetro

```
x: a variável numérica cujo bit será determinado.
n: o bit a ser determinado, começando do 0 para o bit menos significativo.
```

3.6.6 bitClear()

Apaga (escreve um 0 para) um bit de uma variável númerica.

Sintaxe

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

Parâmetro

```
x: a variável numérica cujo bit será apagado.
n: o bit a ser apagado, começando do 0 para o bit menos significativo.
```

3.6.7 bit()

Computa o valor de um bit específico (bit 0 é 1, bit 1 é 2, bit 2 é 4, etc.).

Sintaxe

```
bit(n)
```

Parâmetro

```
n: o bit cujo valor será computado.
```

Retorna

O valor do bit.

3.7 Interrupções externas

3.7.1 attachInterrupt()

Especifica uma chamada de rotina de serviço de interrupção (ISR) para chamar quando ocorre uma interrupção. Substitui qualquer função anterior, que foi anexada à interrupção. A maioria das placas de Arduino tem duas interrupções externas: número 0 (no pino digital 2) e 1 (no pino digital 3). A tabela abaixo mostra os pinos de interrupção em diversas placas.

Placas	int.0	int.1	int.2	int.3	int.4	int.5
Uno, Ethernet	2	3				
Mega2560	2	3	21	20	19	18
Leonardo	3	2	0	1	7	
Due	(veja abaixo)					

O Arduino Due tem poderosas capacidades de interrupção que permitem anexar uma função de interrupção em todos os pinos disponíveis. É possível identificar diretamente o número do pino em `attachInterrupt()`.

Nota

Dentro da função, `delay()` não irá funcionar e o valor retornado por `millis()` não será desenvolvido. Dados Seriais recebidos durante a função podem ser perdidos. É necessário declarar como voláteis todas variáveis que foram modificadas dentro da função. Veja a seção sobte ISRs abaixo para mais informações.

Utilizando interrupções

Interupções são úteis para fazer coisas acontecerem automaticamente em programas de microcontroladores, e pode ajudar a resolver problemas de temporização. Boas tarefas para usar uma interrupção são a leitura de um codificador rotatório, ou monitorar a entrada de um usuário.
Se o objetivo é garantir que o programa sempre conte os pulsos de um codificador rotatório, para que ele nunca perca um pulso, isto tornaria muito complicado escrever um programa para fazer qualquer outra coisa, porque o programa iria precisar consultar constantemente as linhas de sensor para o codificador, a fim de capturar os pulsos quando eles ocorrerem. Outros sensores tem uma dinâmica de interface similar também, como tentar ler um sensor de som que está tentando capturar um ruído, ou um sensor infravermelho (foto-interruptor) tentando capturar uma moeda caindo. Em todas estas situações, usar uma interrupção pode liberar o microcontrolador para capturar outra atividade realizade enquanto não perde a entrada.

Sobre rotinas de serviço de interrupção (Interrupt Service Routines/ISR)

ISRs são tipos especiais de funções que tem algumas limitações únicas que a maioria das outras funções não tem. Uma ISR não pode ter nenhum parâmetro, e não deve retornar nada. Geralmente, uma ISR deve ser o mais curta e rápida quanto possível. Se o código usa múltiplas ISRs, somente uma pode rodar por vez, outras interrupções vão ser ignoradas (desligadas) até a interrupção em questão terminar. Como `delay()` e `millis()` dependem de interrupções, elas

não vão funcionar enquanto uma ISR estiver rodando.
`delayMicroseconds()`, que não depende de interrupções, vai funcionar como esperado. Normalmente, as variáveis globais são usadas para passar dados entre uma ISR e o programa principal. Para ter certeza que variáveis usadas em uma ISR são atualizadas corretamente, as declare como voláteis.

Sintaxe

```
attachInterrupt(interrupt, ISR, mode)
attachInterrupt(pin, ISR, mode)    (Arduino Due somente)
```

Parâmetro

```
interrupt:    o número da interrupção (int)
pin:  o número do pino (Arduino Due somente)
ISR:     o ISR a ser chamado quando uma interrupção ocorre;
esta função não deve ter parâmetros e não deve retornar nada.
Essa função é, algumas vezes, referia como um rotina de serviço de interrupção.
mode:    define quando a interrupção deve  ser acionada. Quatro
constantes são pré-definidas como valores válidos.
LOW para acionar a interrupção sempre que o pino está desligado.
CHANGE para acionar a interrupção sempre que o pino muda o valor.
RISING para acionar quando o pino vai de desligado para ligado.
FALLING para quando o pino vai de ligado para desligado.
A placa Due permite também:
HIGH para acionar a interrupção sempre que o pino está ligado.
(Arduino Due somente)
```

Exemplo

```
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;
}
```

3.7.2 detachInterrupt()

Desliga a interrupção determinada.

Sintaxe

```
detachInterrupt(interrupt)
detachInterrupt(pin)     (Arduino Due somente)
```

Parâmetro

```
interrupt: o número da interrupção a ser desativada
(veja attachInterrupt() para mais detalhes).
pin: o número do pino da interrupção a ser desativada (Arduino Due somente)
```

3.8 Interrupções

3.8.1 interrupts()

Reabilita interrupções (depois delas terem diso desativas por `noInterrupts()`). Interrupções permitem que certas tarefas importantes aconteçam em segundo plano e são ativadas por padrão. Algumas funções não vão funcionar enquanto interrupções são desativadas, e comunicações recebidas podem ser ignoradas. Interrupções podem atrapalhar um pouco o tempo do código no entanto, e podem ser desativadas para seções particulares críticas do código.

Exemplo

```
void setup() {}

void loop()
{
 noInterrupts();
 // crítico, tempo sensível de código aqui.
 interrupts();
 // outro código aqui.
}
```

3.8.2 noInterrupts()

Desativa interrupções (é possível reabilitá-las com `interrupts()`). Interrupções permitem que certas tarefas importantes aconteçam em segundo plano e são ativadas por padrão. Algumas funções não vão funcionar enquanto interrupções são desativadas, e comunicações recebidas podem ser ignoradas. Interrupções podem atrapalhar um pouco o tempo do código no entanto, e podem ser desativadas para seções particulares críticas do código.

Exemplo

```
void setup() {}

void loop()
{
 noInterrupts();
 // crítico, tempo sensível de código aqui.
 interrupts();
 // outro código aqui.
}
```

3.9 Comunicação

3.9.1 Serial

Usada para a comunicação entre a placa do Arduino e o computador ou outros dispositivos. Todas as placas de Arduino tem no mínimo uma porta serial (também conhecida como uma (UART ou USART): Serial. Comunica nos pinos digitais 0 (RX) e 1 (TX) bem como com o computador via USA. Assim, se usadas estas função, não é possível usar também os pinos 0 e 1 para saídas ou entradas digitais.
É possível usar o ambiente serial monitor do Arduino para se comunicar com a placa do Arduino. Clicando no botão serial monitor na barra de ferramentas e selecionando a mesma taxa de transmissão usada na chamada para `begin()`.
O Arduino Mega tem três portas seriais adicionais: Serial1 nos pinos 19 (RX) e 18 (TX), Serial2 nos pinos 17 (RX) e 16 (TX), Serial3 nos pinos 15 (RX) e 14 (TX). Para usar esses pinos para comunicação com um computador pessoal, é necessário um adaptador adicional USB-serial, como eles não estão conectados ao adaptador USB-serial do Mega. Para usá-los para comunicação com um dispositivo serial externo TTL, é necessário conectar o pino TX ao pino RX do dispositivo, o RX ao pino TX do dispositivo, e o ground do Mega ao ground do dispositivo. (Não se conecta esses pinos diretamente a uma porta serial RS232; eles operam em +/- 12V e podem danificar a placa do Arduino.)
O Arduino Due tem três portas seriais TTL 3.3V adicionais: Serial1 nos pinos 19 (RX) e 18 (TX); Serial2 nos pinos 17 (RX) e 16 (TX), Serial3 nos pinos 15 (RX) e 14 (TX). Pinos 0 e 1 são conectados também ao pinos correspondentes do ATmega16U2 USB-TTL Serial chip, que está conectado a porta de depuração USB. Somado a isso, há a porta nativa USB-serial no SAM3X chip, SerialUSB'.
A placa de Arduino Leonardo usa Serial1 para se comunicar via TTL (5V) serial nos pinos 0 (RX) e 1 (TX). Serial é reservada para comunicação USB CDC.

Funções

if (Serial)
available()
begin()
end()
find()
findUntil()

flush()
parseFloat()
parseInt()
peek()
print()
println()
read()
readBytes()
readBytesUntil()
setTimeout()
write()
serialEvent()

3.9.2 Stream

`Stream` é a classe de base para caracteres e streams de bases binárias. Ela não é chamada diretamente, mas chamada sempre que se usa uma função que dependa dela.
`Stream` define funções de leitura no Arduino. Quando se usa qualquer funcionalidade principal que usa `read()` ou métodos similares, é possível seguramente assumir que isso chama uma classe `Stream`. Para funções como `print()`, `Stream` herda da classe `Print`. Algumas das bibliotecas que dependem da `Stream` são:

Serial
Wire
Ethernet Client
Ethernet Server
SD

Funções

available()
read()
flush()
find()
findUntil()
peek()
readBytes()
readBytesUntil()
readString()
readStringUntil()
parseInt()
parsefloat()
setTimeout()

3.10 USB (apenas Arduinos Leonardo e Due)

3.10.1 Teclado e *mouse*

Estas bibliotecas principais permitem que as placas de Arduino Leonardo, Micro, ou Due apareçam como um Mouse e/ou Teclado nativos de um computador conectado.
Um aviso sobre o uso das bibliotecas `Mouse` e `Keyboard`: se a biblioteca `Mouse` ou `Keyboard` está rodando constantemente, isso vai dificultar a programação da placa. Funções como `Mouse.move()` e `Keyboard.print()` vão mover o cursor ou enviar teclas digitadas para um computador conectado e devem somente ser chamadas quando se está pronto para manuseá-las. É recomendado usar um sistema de controle para ativar esta funcionalidade, como um interruptor físico ou somente respondendo a uma entrada específica que pode ser controlada.
Quando se usa a biblioteca `Mouse` ou `Keyboard`, é melhor testar a saída primeiro usando `Serial.print()`. Desse modo, é possível ter certeza que os valores que estão sendo informados são conhecidos. Consulte os exemplos de `Mouse` e `Keyboard` para algumas maneiras de manuseá-las.

Mouse

As funções do mouse permitem ao Leonardo, Micro, ou Due controlar o movimento do cursor do computador conectado. Ao atualizar a posição do cursor, ela é sempre relativa a posição anterior.

Mouse.begin()
Mouse.click()
Mouse.end()
Mouse.move()
Mouse.press()
Mouse.release()
Mouse.isPressed()

Keyboard

As funções do teclado permitem ao Leonardo, Micro, ou Due mandar teclas digitadas para um computador ligado.

Nota

Nem todos possíveis caracteres ASCII, particulamente os não-imprimíveis, podem ser enviados com a biblioteca `Keyboard`. A biblioteca apoia o uso de teclas modificadoras. Teclas modificadoras alteram o comportamento de outra tecla quando pressionada simultaneamente.

Keyboard.begin()
Keyboard.end()
Keyboard.press()
Keyboard.print()
Keyboard.println()
Keyboard.release()
Keyboard.releaseAll()
Keyboard.write()

Apêndice A

Apêndice

A.1 Tabela ASCII

The ASCII code

American Standard Code for Information Interchange

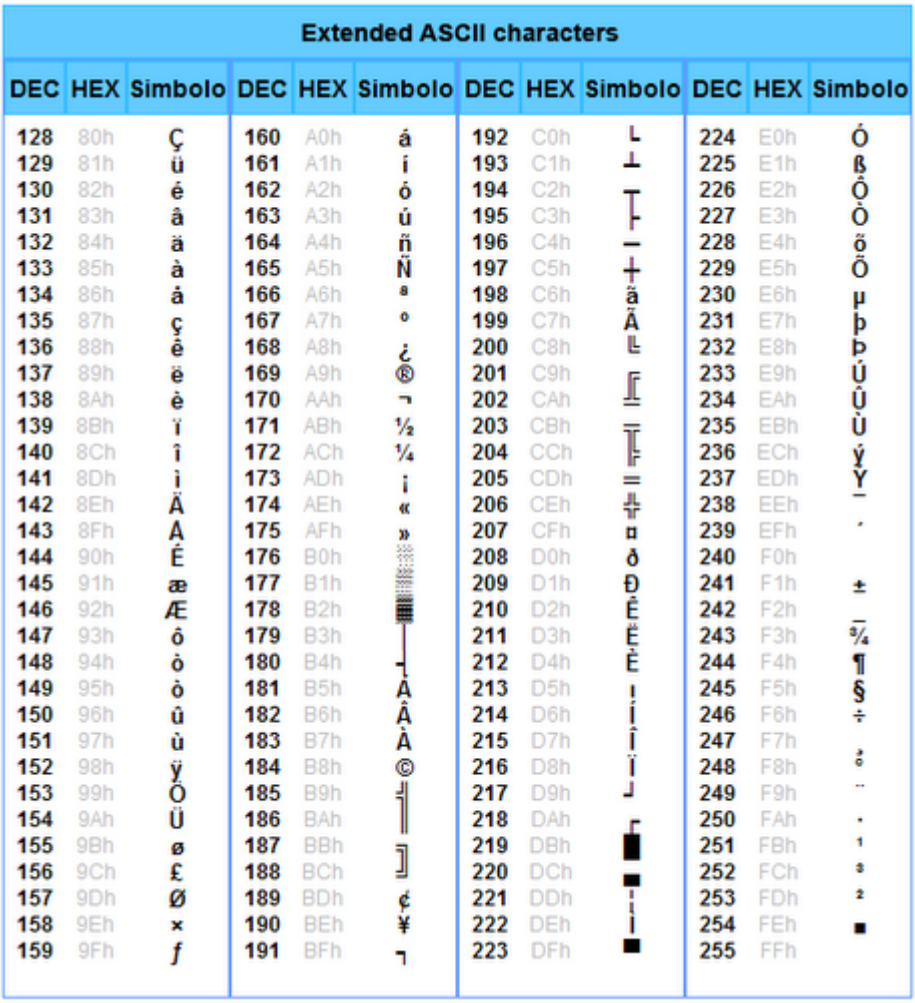
ASCII control characters

DEC	HEX	Simbolo ASCII	
00	00h	NULL	(carácter nulo)
01	01h	SOH	(inicio encabezado)
02	02h	STX	(inicio texto)
03	03h	ETX	(fin de texto)
04	04h	EOT	(fin transmisión)
05	05h	ENQ	(enquiry)
06	06h	ACK	(acknowledgement)
07	07h	BEL	(timbre)
08	08h	BS	(retroceso)
09	09h	HT	(tab horizontal)
10	0Ah	LF	(salto de linea)
11	0Bh	VT	(tab vertical)
12	0Ch	FF	(form feed)
13	0Dh	CR	(retorno de carro)
14	0Eh	SO	(shift Out)
15	0Fh	SI	(shift In)
16	10h	DLE	(data link escape)
17	11h	DC1	(device control 1)
18	12h	DC2	(device control 2)
19	13h	DC3	(device control 3)
20	14h	DC4	(device control 4)
21	15h	NAK	(negative acknowle.)
22	16h	SYN	(synchronous idle)
23	17h	ETB	(end of trans. block)
24	18h	CAN	(cancel)
25	19h	EM	(end of medium)
26	1Ah	SUB	(substitute)
27	1Bh	ESC	(escape)
28	1Ch	FS	(file separator)
29	1Dh	GS	(group separator)
30	1Eh	RS	(record separator)
31	1Fh	US	(unit separator)
127	20h	DEL	(delete)

ASCII printable characters

DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
32	20h	espacio	64	40h	@	96	60h	`
33	21h	!	65	41h	A	97	61h	a
34	22h	"	66	42h	B	98	62h	b
35	23h	#	67	43h	C	99	63h	c
36	24h	$	68	44h	D	100	64h	d
37	25h	%	69	45h	E	101	65h	e
38	26h	&	70	46h	F	102	66h	f
39	27h	'	71	47h	G	103	67h	g
40	28h	(	72	48h	H	104	68h	h
41	29h	)	73	49h	I	105	69h	i
42	2Ah	*	74	4Ah	J	106	6Ah	j
43	2Bh	+	75	4Bh	K	107	6Bh	k
44	2Ch	,	76	4Ch	L	108	6Ch	l
45	2Dh	-	77	4Dh	M	109	6Dh	m
46	2Eh	.	78	4Eh	N	110	6Eh	n
47	2Fh	/	79	4Fh	O	111	6Fh	o
48	30h	0	80	50h	P	112	70h	p
49	31h	1	81	51h	Q	113	71h	q
50	32h	2	82	52h	R	114	72h	r
51	33h	3	83	53h	S	115	73h	s
52	34h	4	84	54h	T	116	74h	t
53	35h	5	85	55h	U	117	75h	u
54	36h	6	86	56h	V	118	76h	v
55	37h	7	87	57h	W	119	77h	w
56	38h	8	88	58h	X	120	78h	x
57	39h	9	89	59h	Y	121	79h	y
58	3Ah	:	90	5Ah	Z	122	7Ah	z
59	3Bh	;	91	5Bh	[	123	7Bh	{
60	3Ch	<	92	5Ch	\	124	7Ch	|
61	3Dh	=	93	5Dh	]	125	7Dh	}
62	3Eh	>	94	5Eh	^	126	7Eh	~
63	3Fh	?	95	5Fh	_			

Figura A.1: Tabela ASCII - parte 1.



Extended ASCII characters

DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
128	80h	Ç	160	A0h	á	192	C0h	└	224	E0h	Ó
129	81h	ü	161	A1h	í	193	C1h	┴	225	E1h	ß
130	82h	é	162	A2h	ó	194	C2h	┬	226	E2h	Ô
131	83h	â	163	A3h	ú	195	C3h	├	227	E3h	Ò
132	84h	ä	164	A4h	ñ	196	C4h	─	228	E4h	õ
133	85h	à	165	A5h	Ñ	197	C5h	┼	229	E5h	Õ
134	86h	å	166	A6h	ª	198	C6h	ã	230	E6h	µ
135	87h	ç	167	A7h	º	199	C7h	Ã	231	E7h	þ
136	88h	ê	168	A8h	¿	200	C8h	╚	232	E8h	Þ
137	89h	ë	169	A9h	®	201	C9h	╔	233	E9h	Ú
138	8Ah	è	170	AAh	¬	202	CAh	╩	234	EAh	Û
139	8Bh	ï	171	ABh	½	203	CBh	╦	235	EBh	Ù
140	8Ch	î	172	ACh	¼	204	CCh	╠	236	ECh	ý
141	8Dh	ì	173	ADh	¡	205	CDh	═	237	EDh	Ý
142	8Eh	Ä	174	AEh	«	206	CEh	╬	238	EEh	¯
143	8Fh	Å	175	AFh	»	207	CFh	¤	239	EFh	´
144	90h	É	176	B0h	░	208	D0h	ð	240	F0h	
145	91h	æ	177	B1h	▒	209	D1h	Ð	241	F1h	±
146	92h	Æ	178	B2h	▓	210	D2h	Ê	242	F2h	‗
147	93h	ô	179	B3h	│	211	D3h	Ë	243	F3h	¾
148	94h	ò	180	B4h	┤	212	D4h	È	244	F4h	¶
149	95h	ò	181	B5h	Á	213	D5h	ı	245	F5h	§
150	96h	û	182	B6h	Â	214	D6h	Í	246	F6h	÷
151	97h	ù	183	B7h	À	215	D7h	Î	247	F7h	¸
152	98h	ÿ	184	B8h	©	216	D8h	Ï	248	F8h	°
153	99h	Ö	185	B9h	╣	217	D9h	┘	249	F9h	¨
154	9Ah	Ü	186	BAh	║	218	DAh	┌	250	FAh	·
155	9Bh	ø	187	BBh	╗	219	DBh	█	251	FBh	¹
156	9Ch	£	188	BCh	╝	220	DCh	▄	252	FCh	³
157	9Dh	Ø	189	BDh	¢	221	DDh	¦	253	FDh	²
158	9Eh	×	190	BEh	¥	222	DEh	Ì	254	FEh	■
159	9Fh	ƒ	191	BFh	┐	223	DFh	▀	255	FFh	

Figura A.2: Tabela ASCII - parte 2.

Modificado de: link ASCII[1] em 12/02/2014.

[1] http://www.theasciicode.com.ar/american-standard-code-information-interchange/ascii-codes-table.gif

Referências Bibliográficas

[1] Referência do *website* oficial do Arduino. http://arduino.cc/en/Reference/HomePage. Accessado em: 12/02/2014.

[2] Waldemar Celes Filho, Renato Cerqueira, and Joé Lucas Rangel. *Introdução a estruturas de dados: com técnicas de programação em C*. Elsevier, 11 edition, 2004.

[3] Neal S. Widmer and Ronald J. Tocci. *Sistemas Digitais - Princípios e Aplicações*. Prentice Hall - Br, 11 edition, 2011.