Arduino Programming Notebook

Brian W Evans

Edição brasileira

Tradução e adaptação:

Almir Mendes Ventura

www.omecatronico.com.br

Ver. 1.0 de 25/04/2014

Dados do livro original:

Arduino Notebook: A Beginner's Reference Written and compiled by
Brian W. Evans

With information or inspiration taken from:
http :// www . arduino . cc
http :// www . wiring . org . co
http://www.arduino.cc/en/Booklet/HomePage
http :// cslibrary . stanford . edu /101/

Including material written by:
Massimo Banzi
Hernando Barragan
David Cuartielles
Tom Igoe
Todd Kurt
David Mellis and others

Published:
First Edition August 2007

Conteúdo

Prefacio

O propósito do autor original deste livro foi criar um pequeno manual de consulta rápida sobre os comandos básicos e a sintaxe da linguagem de programação do Arduino. Para mantê-lo simples o autor retirou alguns assuntos mais complexos, e tornou este livro mais indicado para os iniciantes em Arduino. Esta decisão tornou este um livro uma abordagem genérica para aprender Arduino na sua forma padrão e básica sem ênfase em assuntos muito específicos como arrays ou comunicação serial avançada.

Para avançar nos conceitos mais complexos, podem ser consultadas outras fontes incluindo páginas da web, livros, workshops e cursos.

Começando com a estrutura básica da linguagem Arduino, que foi derivada da linguagem C, este livro continua com a descrição da sintaxe da maioria dos elementos da linguagem e ilustrando com exemplos e fragmentos de código. Ele inclui várias funções da biblioteca padrão bem como um apêndice que inclui amostras de esquemas de ligação de periféricos e programas básicos.

Esta tradução para o português foi realizada por Almir Mendes Ventura para que iniciantes em Arduino pudessem ter uma apostila que fosse simples e em português. Como muitos possuem certa dificuldade em ler textos em inglês esta tradução deverá servi-los bem.

Este livro é uma combinação de esforços de toda uma comunidade, incluindo a www.arduino.cc que sem eles não existiria o Arduino, bem como o autor original **Brian W Evans** que publicou seu trabalho de forma aberta, bem como todos os autores os quais Brian teve como base.

A estrutura de um programa

Sketch (Esboço)

A estrutura basica da linguagem de programação Arduino é bastante simples e roda basicamente em duas partes. Estas duas partes obrigatórias, ou funções, delimitam blocos de comandos.

```
void setup()
{
  comandos;
}

void loop()
{
  comandos;
}
```

Onde **setup**() é a configuração ou inicialização e **loop**() é onde fica o programa que será repetido ciclicamente. Ambas as funções são necessárias para que o programa funcione.

A função do setup pode conter as declarações de quaisquer variáveis no início do programa. Ela é a primeira função a ser executada no programa e é executada apenas uma vez. Geralmente é utilizada para escolher o modo dos pinos bem como inicializar a comunicação serial.

A função loop é executada logo em sequência e inclui o código que será executado continuamente – lendo entradas, ligando ou desligando saídas, etc. Esta função é a principal em todos os programas Arduino e é ela quem executa a maior parte do trabalho.

setup()

A função **setup**() é chamada apenas uma única vez quando seu programa inicia. Use-a para inicializar os pinos ou inicializar a serial. Ela deve ser incluída em todo programa mesmo que não tenha nenhum comando para ser executado.

```
void setup()
{
  pinMode(pino, OUTPUT);            //configura “pino” como saída
}
```

Loop()

Depois de chamar a função **setup**(), a função **loop**() faz exatamente o que o nome sugere, fica num "loop" repetindo sem parar, permitindo que o programa mude, responda e controle a placa do Arduino.

```
void loop()
{
digitalWrite(pin, HIGH);          //faz o 'pin' ligar(seta o pino)
delay(1000);                      //pausa durante 1 segundo
digitallWrite(pin, LOW);          //faz o 'pin' desligar(zera o pino)
delay(1000);                      //pausa durante 1 segundo
}
```

Funções

Uma função é um bloco de código que possui um nome e um bloco de instruções que são executadas quando a função for chamada. As funções void **setup**() e void **loop**() já foram discutidas e outras funções nativas serão discutidas mais na frente.
Funções próprias podem ser escritas para realizarem tarefas repetitivas e reduzir a confusão em um programa. As funções são criadas primeiramente declarando seu tipo. Este tipo será o tipo de valor que será retornado pela função, como por exemplo um 'int' para uma função do tipo inteira. Se nenhum valor será retornado então a função será do tipo void. Depois do tipo, escreva o nome que você escolher para chamar a função e logo em seguida, entre parênteses, coloque os parâmetros que serão passados para a função.

```
tipo nomedafunção(parâmetros)
{
  Comandos;
}
```

A função do tipo inteira a seguir chamada delayVal() é usada para obter um valor de delay em um programa atravé da leitura do valor de um potenciometro. Primeiro ela declara uma variável local v, depois guarda em v o valor do potenciômetro, retornado pela função analogRead(). Este valor varia entre 0 e 1023, então logo em seguida ela divide por 4 obtendo assim um valor entre 0 e 255. Finalmente ela retorna esse valor para o programa principal.

```
Int delayVal()
{
  Int v;                          //cria uma variável temporária v
  v = analogRead(pot);            //lê o valor do potenciômetro
  v /= 4;                         //converte 0-1023 para 0-255
  return v;                       //retorna o valor final
}
```

{ } chaves

As chaves definem o inicio e o fim de um bloco de uma função ou bloco de comandos como na função void **loop***()* e para os comandos *for* e *if.*

```
tipo função()
{
  comandos;
}
```

Uma chave de abertura { sempre deve ser seguida por uma chave de fechamento }. Isto geralmente se referem ao balanceamento de chaves. Chaves desbalanceadas podem levar a erros críticos, impenetráveis e confusos do compilador levando a uma tarefa árdua para descobrir o erro em um programa grande.
O ambiente de desenvolvimento do Arduino inclui uma ferramenta interressante para checagem do balanceamento das chaves. Apenas selecione uma chave, ou clique exatamente antes de uma chave, e a chave que complementa a que você selecionou ficará em destaque.

; Ponto e vírgula

O ponto e vírgula devem ser usados para terminar um comando e separar os elementos de um programa. O ponto e vírgula também é usado para separar os parâmetros em um loop *for*.

```
Int x = 13;              //declara a variável x e atribui a ela o valor 13
```

Nota: Esquecer de terminar um comando com um ponto e virgula resultará em um erro na compilação. O erro acusado pelo compilador pode ser obvio e mostrar que está faltando um ponto e virgula ou não. Se um erro sem sentido ou ilógico aparecer, uma das primeiras coisas a fazer é procurar um ; que está faltando próximo da linha que o compilador está reclamando.

/*...*/ blocos de comentários

Blocos de comentários, ou comentários de varias linhas, são áreas de texto ignoradas pelo programa e são usadas para textos de descrição de código ou comentários que ajudarão os outros a entenderem partes do programa. Eles são iniciados por /* e terminam com */ e podem ocupar varias linhas.

```
/*  este é um bloco de comentário
   não esqueça de fechar o bloco de comentários
   eles precisam ser fechados também!
*/
```

Por causa dos comentários serem ignorados pelo programa e não consumirem nenhum espaço, eles devem ser usados de forma generosa e podem ser usados para "comentar" blocos de código que são usados apenas para fins de testes.

Nota: Mesmo sendo possível colocar dentro de um comentário de bloco um comentário de linha, anexar um segundo bloco de comentário não é permitido.

// comentários de linha

Comentários de linha começam com // e terminam com a próxima linha de código. Da mesma forma que os comentários de bloco, estes também são ignorados pelo programa e não ocupam espaço algum da memória do Arduino.

```
// este é um comentário de linha
```

Comentários de linha são usados geralmente após um comando para prover de mais informação do que este comando está fazendo ou para deixar um lembrete para futuramente.

Variáveis

Uma variável é uma maneira de nomear e guardar um valor numérico para uso posterior pelo programa. Como o nome delas já sugerem, variáveis são números que podem ser continuamente modificados e são o oposto das *constantes* as quais nunca mudam seus valores. Uma variável precisa ser declarada e opcionalmente receberá o valor que precisa ser guardado. O código a seguir declara uma variável chamada varEntrada e então armazena nela o valor obtido pelo pino analógico 2:

```
Int varEntrada = 0;              //declara a variável e
                                 //armazena o valor 0
varEntrada = analogRead(2);      //armazena o valor retornado
                                 //pelo pino analógico 2
```

'varEntrada' é a variável em si. A primeira linha declara que a variável armazenará um 'int', que é a abreviação para inteiro. A segunda linha armazena na variável o valor do pino analógico 2. Com isto o valor do pino 2 ficará disponível em qualquer parte do código.

Uma vez que a variável foi inicializada ou recebeu outro valor, você pode testar seu valor para ver se ele condiz com certas condições ou você pode usar o valor diretamente. Como exemplo para ilustrar 3 operaçoes úteis com variáveis, o código a seguir testa se a variável varEntrada é menor que 100, se verdadeiro ela vai atribuir 100 a variável e logo após faz um delay baseado na variável varEntrada que agora tem no mínimo 100:

```
if(varEntrada < 100)          //testa de varEntrada é menor que 100
```

```
{
  varEntrada = 100;                    //se verdade atribui 100 a variável
}
delay(varEntrada);                     //usa a variável como parâmetro do delay
```

Nota: variáveis devem possuir nomes sugestivos para facilitar a leitura do código. Nomes de variáveis como tiltSensor ou pushButton facilitam a vida do programador e de quem quer que esteja lendo o código para entender o que a variável representa. Variáveis com nomes feito var ou valor, por outro lado, podem dificultar um pouco o entendimento do programa e são utilizadas aqui apenas como exemplos. Uma variável pode ter qualquer nome desde que não seja igual a nenhum nome ou função da linguagem Arduino. Obrigatoriamente uma variável deve começar com uma letra ou underscore ("_"). O restante pode ser letras de A a Z, maiúsculas, minúsculas, números e o underscore;
Ex: a; num; essa_e_uma_variavel; tecla6; tecla_6;
Não podemos ter variáveis com nomes tipo: if, int, break, 6tecla ...etc.

Declaração de Variável

Todas as variáveis tem que ser declaradas antes de serem usadas. Declarar uma variável significa definir seu tipo de armazenamento como int, long, float, etc., definindo um nome para variável e opcionalmente atribuir um valor inicial a ela. Isto precisa ser feito apenas uma vez no programa enquanto que o valor armazenado por ela pode ser alterado a qualquer momento usando aritmética ou varias outras formas.

O exemplo a seguir declara que varEntrada será do tipo int, ou seja, do tipo inteiro e que seu valor inicial será igual a zero. Esta é considerada uma declaração simples.

```
Int varEntrada = 0;
```

Uma variável pode ser declarada em vários lugares em um programa e onde ela será feita determina que partes de um programa podem utilizá-la.

Escopo de uma variável

Uma variável pode ser declarada no inicio de um programa antes de void **setup**(), localmente dentro de funções e algumas vezes até dentro de parâmetros de funços como na estrutura de repetição *for*. Onde a variável é declarada determina qual será seu escopo, ou abilidade de certas partes do programa poderem utilizá-la.

Uma variável global é a variável que pode ser vista e usada por qualquer função ou parte do programa. Este tipo de variável é declarada no inicio do programa, antes da função **setup**().

Uma variável local é a variável definida dentro de uma função ou como parte de um loop. Ela é apenas visível e somente pode ser usada dentro da função em que ela foi declarada. Com isso é possível ter uma ou mais variáveis com mesmo nome em diferentes partes de um programa que contem diferentes valores. Garantir que apenas

uma função tem acesso a suas próprias variáveis simplifica os programas e reduzem bastante a possibilidade de erros.
O exemplo a seguir mostra como declarar alguns tipos diferentes de variáveis e demonstra a visibilidade de cada uma.

```
int valor;                    // 'valor' é visível para qualquer função (global)

void setup()
{
  //nenhum  setup é necessário neste exemplo
}

void loop()
{
  for(int i=0; i<20; )
  {
    i++;                      // 'i' é visível apenas dentro do for
  }
  float f;                    // 'f' é visível apenas dentro de loop
}
```

Tipos de Variáveis

byte

Byte armazena um valor numeric de 8 bits sem ponto decimal. Ele tem um alcance de 0-255.

```
byte algumaVariavel = 180;   //declara "algumaVariavel" como tipo byte
```

int

Inteiros são os tipos primários para armazenamento de números sem o ponto decimal e armazenam um valor de 16bit e tem um alcance de 32767 até -32768.

```
int algumaVariavel = 1500;    //declara "algumaVariavel" como tipo inteiro
```

Nota: variáveis inteiras vão reverter seu valor se forem forçadas a passar do seu valor máximo ou mínimo através de uma atribuição ou comparação. Por exemplo, se x = 32767 e um comando seguinte adiciona 1 a x, x=x+1 ou x++, x vai reverter seu valor para -32768.

long

São inteiros com um alcance estendido para 32bit, e como são inteiros, não possuem casa decimal. Seu alcance vai de 2.147.483.647 até -2.147.483.648.

```
long algumaVariavel = 90000;          //declara "algumaVariavel" como tipo long
                                      //e inicializa com o valor 90000
```

float

Um tipo de variável para números com casas decimais (tambem chamado ponto flutuante). Pontos flutuantes possuem uma resolução maior que os inteiros e são armazenados com 32bit e possuem um alcance de 3,4028235E+38 até -3,4028235E+38.

```
float algumaVariavel = 3.14;  //declara "algumaVariavel" como tipo float
                              // e inicializa com o valor 3.14
```

Nota: Números de ponto flutuante não são exatos, e podem levar a resultados estranhos quando comparados. Tambem cálculos matemáticos envolvendo ponto flutuante são muito lentos quando comparados com cálculos envolvendo apenas inteiros. Então evite quando puder os cálculos com ponto flutuante.
Nota do tradutor: Quando desejar comparar se dois valores ponto flutuante são iguais, você deve fazer uma comparação com um "erro", tão pequeno quanto se desejar, como exemplo ao invés de fazer o teste x==y, devemos testar se o módulo da diferença é menor que um erro que escolhemos, ou seja, abs(x-y) < 0.001.

Arrays

Um array é uma coleção de valores que são acessados com um numero de indexação. Qualquer valor no array pode ser chamado através do nome do array seguido do numero de indexação. Arrays começam com índice zero. Um array precisa ser declarado e opcionalmente terem seus valores inicializados antes de serem utilizados.

```
int meuArray[] = {valor0, valor1, valor2...}
```

Da mesma forma é possível declarar um array com seu tipo e tamanho e somente mais na frente atribuir os respectivos valores em seus respectivos índices:

```
int meuArray[5];          //declara um array de inteiros com 5 posições (de 0 a 4)
meuArray[3] = 10;         //atribui ao 4º item o valor 10
```

Para recuperar um valor de um array, utilize uma variável para receber o valor e chame o array colocando o índice do item que você deseja recuperar:

```
x = meuArray[3];          //x agora tem o valor 10
```

Arrays geralmente são utilizados em loops for, onde o contador de interações também é utilizado como indexador para cada índice do array. O exemplo a seguir usa um array para fazer um LED tremular (flicker). Usando um loop for, o contador 'i' inicia com 0, o valor do contador será usado como índice do array flicker[] para acesar o item correspondente, que nesse instante é o item 0 que possui o valor 180. Este valor é repassado para o PWM no pino 10, faz uma pausa de 200ms e então recomeça o ciclo do próximo numero do contador.

```
int ledPin = 10;                              //LED no pino 10
byte flicker[] = {180,30,255,200,10,90,150,60};   //array com 8 valores diferentes

void setup()
{
  pinMode(ledPin, OUTPUT);          //configura ledPin para ser saída
}

void loop()
{
  for(int i=0; i<7; i++)            //loop com a mesma quantidade de itens no array
  {
    analogWrite(ledPin, flicker[i]);  //escreve o valor do item do array no PWM do LED
    delay(200);                       //pausa por 200ms
  }
}
```

Aritmética

Operadores aritméticos incluem adição, subtração, multiplicação e divisão. Eles retornam a soma, diferença, produto ou quociente (respectivamente) de dois operandos.

```
y = y +3;
x = x – 7;
i = j * 6;
r = r / 5;
```

A operação é conduzida pelo tipo dos operandos, por exemplo, 9 / 4 tem resultado 2 ao invés de 2.25 desde que 9 e 4 são inteiros e são incapazes de usar casas decimais. Isto também significa que o resultado da operação pode ultrapassar os limites que podem ser armazenados pelo tipo da variável e causar problemas como um overflow.
Se os operandos são de tipos diferentes, o tipo maior será usado para o cálculo. Por exemplo se um dos números (operandos) é do tipo float e o outro um inteiro, será usada matemática de ponto flutuante(float).
Escolha variáveis que são grandes o suficiente para armazenar os maiores resultados possíveis de seus cálculos. Conheça em que ponto suas variáveis vão reverter e o que acontece quando vão na direção oposta, ex.: (0 - 1) ou (0 - - 32768). Para cálculos que requerem frações, use variáveis tipo float, mas tenha em mente as suas desvantagens: ocupam muita memória e possuem computação lenta.
Nota: Use o conversor de tipos ex.: (int)meuFloat para converter um tipo de variável em outro em qualquer momento. Por exemplo, i = (int)3.6 vai armazenar em i o valor 3.

Atribuições Compostas

Atribuições Compostas são as combinações de uma operação aritmética com a atribuição a uma variável. Estes operadores são comumente encontrados em repetições for como mostrado anteriormente. Os operadores mais comuns incluem:

```
x++     //o mesmo que x=x+1, ou incrementa x de +1
x- -    //o mesmo que x=x-1, ou decrementa x de -1
x+=y    //o mesmo que x=x+y, ou incrementa x de +y
x-=y    //o mesmo que x=x-y, ou decrementa x de –y
x*=y    //o mesmo que x=x*y, ou multiplica x por y
x/=y    //o mesmo que x=x/y, ou divide x por y
```

Nota: Por exemplo, x*=3 vai triplicar o valor antigo de x e armazenar o resultado em x.

Operadores de comparação

Comparações de uma variável ou constante com outra são geralmente utilizadas em comandos IF para testar se uma condição especifica é verdadeira. No próximos exemplos, ?? será usado para indicar qualquer uma das seguintes condições:

```
x == y
x != y
x <  y
x >  y
x <= y
x >= y
```

Operadores Lógicos

Operadores lógicos são o meio comum de comparar duas expressões e retornarem um TRUE ou FALSE dependendo do operador. Existem 3 tipos de operadores lógicos, AND(E), OR(OU) e NOT(NÃO), que são utilizados nos comandos:

O E lógico (AND):

```
if (x > 0 && x < 5)       //é verdadeiro apenas se ambas expressões forem verdadeiras
```

O OU lógico (OR):

```
if(x > 0 || y > 0)        //é verdadeiro de pelo menos uma é verdadeira
```

O NÃO lógico (NOT)

```
if(! x > 0)               //é verdadeiro apenas se a expressão for falsa
```

Constantes

A linguagem Arduino possui alguns valores predefinidos, os quais são chamados constantes. Elas são utilizadas para tornarem os programas fáceis de ler. Constantes são classificadas em grupos.

True/False (verdadeiro/Falso)

Estes são constantes do tipo booleano que definem níveis lógicos. FALSE é facilmente definido como 0 (zero) enquanto que TRUE é definido como 1, mas pode ser também qualquer coisa exceto zero. Então na lógica booleana, -1, 2 e -200 tambem são considerados como TRUE.

```
if (b == TRUE)
{
  comandos;
}
```

High/Low (Alto/Baixo)

Estas constantes definem os níveis dos pinos como HIGH ou LOW e são utilizados quando se lê ou se escreve em pinos digitais. HIGH é definido como nível lógico 1, ligado ou 5volts enquanto que LOW é o nível lógico 0, desligado ou 0volts.

```
digitalWrite(13, HIGH);          //faz o pino 13 ligar (ficar com 5volts)
```

INPUT / OUTPUT (Entrada/Saída)

Constantes utilizadas em conjunto com a função pinMode() que define o modo como um pino digital se comportará, se será entrada(INPUT) ou saída(OUTPUT).

```
pinMode(13, OUTPUT);          //define que o pino 13 será utilizado como saída
```

Controle de Fluxo

if (comando SE)

Os comandos if testam se uma certa condição foi atingida, como um valor analógico acima de um certo numero, e executa os comando dentro das chaves { } se o a condição for verdadeira. Se for falsa o programa ignora e pula todo o bloco do if e continua.

```
if(algumaVariavel ?? valor)
{
  façaAlgumaCoisa;
}
```

O exemplo acima compara algumaVariavel com outro valor, que pode ser tanto uma variável como uma constante. Se a comparação, ou condição, em parênteses for verdadeira, os comandos dentro das chaves serão executados. Se não, o programa não executa nada entre as chaves, e continua exatamente após o fechamento da chave }.
Nota: tenha cuidado em acidentalmente usar '=', como em if(x = 10), embora tecnicamente válido, faz uma atribuição do valor 10 a variável x o que sempre resulta verdadeiro. Ao invés disso use '==', como em if(x == 10), que nesse caso testa se a variável x tem valor igual a 10 ou não. Pense em '=' como "recebe o valor" oposto ao '==' que seria "tem valor igual a".

if...else (SE ...SENAO...)

If...else permitem que comandos sejam executados quando a condição de teste não foi atingida, seria o "caso contrário". Por exemplo, se você quer testar uma entrada digital e fazer uma coisa se a entrada for para HIGH ou fazer outra coisa quando ela for para nível LOW, você escreveria dessa forma:

```
if(inputPin == HIGH)
{
   façaComandosA;            //somente executa esse bloco se o pino for 5volt (HIGH)
}
else
{
   façaCoamndosB;            //somente executa esse bloco se o pino for 0volt (LOW)
}
```

else também pode preceder outro teste if, então testes múltiplos e mutuamente exclusivos podem ser feitos de uma vez. É possível colocar um número ilimitado de else concatenados. Lembrando que apenas um bloco de comandos será executado dependendo das condições do teste.

```
if(inputPin < 500)
{
   executeComandosA;
}
else if(inputPin >= 1000)
{
   esecuteComandosB;
}
else
{
   executeComandosC;
}
```

Nota: o comando if simplesmente testa se uma condição dentro dos parênteses é verdadeira ou falsa. Esta condição pode ser qualquer uma válida de linguagem C, como no primeiro exemplo, if(inputPin == HIGH). Neste exemplo, o comando if apenas checa se o pino em questão possui nível lógico alto (+5volts).

For (repetir n vezes)

O comando for é utilizado para repetir um bloco de comandos delimitados pelas chaves { } por um número especificado de vezes. Um contador de incrementos é sempre utilizado para incrementar e terminar o loop. É formado por três parâmetros separados por ponto e virgula(;) que ficam no cabeçalho do comando.

```
for(inicialização ; condição ; expressão)
{
   executeComandos;
}
```

A inicialização de uma variável local, ou contador de incrementos, acontece no inicio e apenas uma vez. Cada vez através do loop, a “condição” é testada. Se a condição continuar verdadeira, uma nova repetição será feita, ocasionando a “expressão” e também os comandos serem executados novamente. Quando a condição for falsa o loop termina.
O exemplo a seguir inicia o inteiro i com 0, testa se ele ainda é menor que 20 e se verdadeiro, incrementa i de 1 e executa os comandos entre as chaves:

```
for(int i=0; i < 20; i++ )        //declara i, testa se é menor que 20 e incrementa i de 1
{
   digitalWrite(13, HIGH);        //liga o pino 13
   delay(250);                    //pausa por ¼ de segundo
   digitalWrite(13, LOW);         //desliga o pino 13
   delay(250);                    //pausa por ¼ de segundo
}
```

Nota: o comando for na linguagem C é muito mais flexível que outros loops for de outras linguagens, incluindo o BASIC. Nenhum ou os três parâmetros do cabeçalho

podem ser omitidos, apesar dos ponto e vigula(;) serem necessários. Adicionalmente os parâmetros de inicialização, condição e expressão podem ser qualquer expressão valida em C e com variáveis não relatadas. Estes tipos não usuais de parâmetros podem fornecer soluções para alguns casos raros de programação.

While (repetir enquanto...)

A repetição while vai repetir continuamente e indefinidamente até que a expressão entre parênteses se torne falsa. Alguma coisa tem que mudar a variável de teste, ou o loop while nunca vai ter fim. Isto pode ser em seu código, uma variável incrementada ou uma condição externa como o teste de um sensor.

```
while(algumaVariavel ?? valor)
{
   executeComandos;
}
```

O exemplo a seguir testa se 'algumaVariavel' é menor que 200 e se verdadeiro executa os comandos entre as chaves { } e vai continuar repetindo até que 'algumaVariavel' não seja mais menor que 200.

```
while(algumaVariavel < 200)        //testa se é menor que 200
{
   executeComandos;                //executa os comandos que estão ente as chaves
   algumaVariavel++;               //incrementa a variável de +1
}
```

do...while(executar e repetir enquanto...)

A repetição 'do' é um loop comandado no final mas que funciona da mesma maneira que a repetição while, com a pequena diferença de que o teste de condição do loop fica no final ao invés de no topo. Com isso a repetição será executada pelo menos uma única vez.

```
do
{
   executeComandos;
} while(algumaVariavel ?? valor);
```

O exemplo a seguir atribui o valor de retorno da função readSensors() a variável 'x', pausa por 50ms e depois, repete indefinidamente até que 'x' não seja mais inferior a 100.

```
do
{
   x = readSensors( );          //atribui o valor de retorno de readSensors() a x
   delay(50);                   //pausa por 50 milissegundos
} while( x < 100 );             //repete se x for inferior a 100
```

Entradas e Saídas Digitais (Digital i/o)

pinMode(pino, modo)

Utilizado em void setup() para configurar um pino especificado e torna-lo INPUT (entrada) ou OUTPUT (saída). Exemplo:

```
pinMode(10, OUTPUT);          //faz o pino 10 funcionar como saída.
```

O padrão do Arduino para pinos digitais é serem entradas (INPUT), com isso não é preciso explicitamente declarar que o pino será uma entrada digital com pinMode(). Pinos configurados como INPUT ficam em um estado de "alta impedância".
Nos chips AtMega existem resistores de pullup de 20KΩ que podem ser acessados por software. Estes resistores inclusos podem ser acessados da seguinte maneira:

```
pinMode(pino, INPUT);         //torna 'pino' uma entrada
digitalWrite(pino, HIGH);     //habilita os resistores de pullup
```

Resistores de pullup normalmente são utilizados para conectar chaves por exemplo. Observe que no exemplo acima o comando não converte o 'pino' em OUTPUT (saída), é meramente um método para ativar os resistores internos de pullup.
Pinos configurados como OUTPUT são ditos que ficam em baixa impedância e podem prover até 40mA(miliamperes) de corrente para outros dispositivos ou circuitos. Isso é corrente suficiente para ligar um LED com forte brilho (não esqueça de ligar um resistor em série com o LED ou você queimara o Arduino), mas não é corrente suficiente para ligar a maioria dos reles, solenóides ou motores.
Curto circuitos nos pinos do Arduino e correntes excessivas podem danificar ou destruir o pino, ou até mesmo destruir o chip AtMega inteiro. É sempre uma boa idéia conectar um resistor de 470Ω ou 1kΩ em série com qualquer pino que seja utilizado como saída (OUTPUT).

digitalRead(pino)

Lê o valor(estado lógico) do pino configurado como entrada digital e obtém como resultado HIGH(1) ou LOW(0). O pino pode ser especificado através de variável ou através de constante 0 até 13 no caso do Arduino UNO.

```
valor = digitalRead(pino);    //'valor' recebe 0 ou 1 da leitura do pino 'pino'
```

digitalWrite(pino, Valor)

Escreve o nivel lógico HIGH ou LOW (liga ou desliga) em um determinado pino digital. O pino pode ser especificado através de variável ou através de constante(0 -13) no caso do Arduino UNO.

```
digitalWrite(pino, HIGH);     //seta o 'pino' (liga ou coloca 5volts no pino)
```

O exemplo a seguir lê o estado de um botão conectado ao pino 7(configurado como entrada digital) e liga um LED conectado ao pino 13 (configurado como saída digital) quando o botão for pressionado.

```
int led = 13;           //declara a variável led e atribui o valor 13 (pino do led)
int botao = 7;          //declara a variável botao e atribui o valor 7 (pino do botão)
int valor = 0;          //variável para armazenar temporariamente o valor lido

void setup()
{
  pinMode(led, OUTPUT);   //faz o pino 13 ser saída
  pinMode(botao, INPUT);  //faz o pino 7 ser entrada
}

void loop()
{
valor = digitalRead(botao);   //armazena em 'valor' o estado lógico(1 ou 0) do pino 7
digitalWrite(led, valor);     //faz o pino 'led' ter mesmo estado lógico que o botao
}
```

Entradas e Saídas Analógicas (analog i/o)

analogRead(pino)

Lê o valor de um pino analógico e isso com uma resolução de 10bits. Esta função apenas funciona nos pinos analógicos (0-5 no Arduino UNO). O retorno dessa função é um inteiro que varia de 0 (0volts) até 1023 (5volts). Se você dividir 5volts por 1024 niveis encontrará aproximadamente 0,00488 volts por incremento, sendo uma resolução boa para muitos casos.

```
x = analogRead(pino);       //'x' recebe valor entre 0 e 1023 proporcional a
                            //tensão do 'pino'
```

Nota: Pinos analógicos ao contrario dos pinos digitais, não precisam ser declarados como INPUT ou OUTPUT.

analogWrite(pino, valor)

Escreve um valor pseudo-analógico utilizando o recurso de hardware de Pulse width Modulation(PWM) para um pino marcado com esse recurso(possui um ~ ao lado do nome). Nos novos Arduinos com AtMega168 ou AtMega328, este recurso está disponível nos pinos 3,5,6,9,10 e 11. Arduinos antigos com Atmega8 apenas possuem o recurso nos pinos 9,10 e 11.
O valor pode ser declarado através de variável ou de constante e fica entre 0 e 255.

```
analogWrite(pino, valor);   //escreve 'valor' no 'pino'
```

Um valor 0 gera uma tensão estável de 0volts no pino especificado; um valor de 255 gera uma tensão estável de 5volts no pino especificado. Para valores entre 0 e255, o pino vai ficar alternando rapidamente entre 0 e 5volts – quanto maior o valor, mais tempo o pino fica em HIGH(5volts)e proporcionalmente mais tensão no pino.

Por exemplo, um valor de 64 vai passar 75% do tempo em 0volt e 25% do tempo em 5volts; um valor de 128 vai passar metade do tempo em 0volt e a outra metade em 5volts; um valor de 192 vai passar 25% do tempo em 0volt e 75% do tempo em 5volts.

Como o PWM é gerado em hardware, o pino vai gerar e manter uma onda estável após a chamada da função analogWrite(). Esta onda somente será alterada caso uma nova chamada a analogWrite() ou seja feita no mesmo pino uma chamada as funções digitalRead() ou digitalWrite().

Nota: os pinos analógicos ao contrario dos pinos digitais, não precisam ser previamente declarados como INPUT ou OUTPUT.
O código a seguir lê um valor analógico de um pino analógico, converte o valor dividindo por 4, e então, coloca em um pino de saída PWM o valor.

```
int led = 10;    //LED com um resistor em serie de 220Ω no pino 10
int pin = 0;     //potenciometro no pino analógico 0
int valor;       //variável temporária para calculos

void setup()    //não será necessário o setup nesse exemplo
{
}
void loop()
{
valor = analogRead(pin);      //'valor' recebe a leitura do pino analogico
valor /= 4;                   //converte 0-1023 para 0-255
analogWrite(led, valor);      //gera o sinal PWM pro led
}
```

Controle de tempo

delay(ms)

Pausa o programa pelo tempo em milissegundos especificado entre os parênteses, sabendo que 1000ms correspondem a 1 segundo.

```
delay(1000);    //espera por 1 segundo
```

millis()

Retorna o numero em milissegundos desde que a placa Arduino iniciou a rodar o programa atual. O valor é um unsigned long.

```
x = millis();              //faz 'x' armazenar o valor atual de millis()
```

Nota: este valor vai ter overflow(voltar para o zero) após aproximadamente 50 dias.

Math - Funções Matemáticas

min(x,y)

Calcula o mínimo valor entre dois números de qualquer tipo e retorna o menor dos dois.

```
valor = min(valor, 100);        //armazena em valor o minimo entre ele mesmo e 100
                                //garantindo assim que valor nunca ultrapasse 100.
```

max(x,y)

Calcula o máximo entre dois numeros de qualquer tipo e retorna o valor do maior deles.

```
valor = max(valor, 100);        //armazena em valor o máximo entre ele mesmo e 100
                                //garantindo assim que valor não fique abaixo de 100
```

Nota do tradutor: existem várias outras funções matemáticas como map(), pow(), sqrt() ou de trigonometria como o sin(), cos(), tan(). Para mais informações consulte o site arduino.cc e acesse "Reference".

Random - Funções Randômicas

randomSeed(seed)

Inicializa com um valor, a semente, como ponto de partida para a função random().

```
randomSeed(valor);          //faz 'valor' ser a semente do random
```

random(maximo) ou random(minimo, maximo)

A função randômica permite retornar um número pseudo-randômico com um alcance especificado por pelos valores mínimo e máximo.

```
valor = random(100,200);    //valor armazena um numero aleatório
                            // que fica entre 100 e 200
```

Nota: usar a função random() apenas depois de usar a função randomSeed().

O exemplo abaixo cria um valor aleatório entre 0 e 255 e depois gera um PWM com este mesmo valor em um pino que possui a capacidade de PWM:

```
int randNumber;      //variável para guardar o valor aleatório
int led = 10;        //LED com resistor de 220Ω no pino 10

void setup()         //não foi necessário o setup neste exemplo
{
}

void loop()
{
  randomSeed( millis() );          //faz o retorno de millis() ser a semente
  randNumber = random(255);        //numero randômico entre 0 e 255
  analogWrite(led, randNumber);    //gera o PWM com o valor gerado anteriormente
  delay(500);                      //pausa por meio segundo
}
```

Serial

Serial.begin(velocidade)

Abre a porta de comunicação serial e configura a velocidade de comunicação (baud rate). A velocidade típica é de 9600baud embora outras velocidades sejam suportadas.

```
void setup()
{
  Serial.begin(9600);    //inicia a serial e configura 9600baud de velocidade
}
```

Nota: enquanto a comunicação serial estiver sendo utilizada, o pino 0(RX) e o pino 1(TX) não poderão ser utilizados. Se o seu projeto vai manter a comunicação serial então estes pinos não poderão ser usados de forma alguma.

Serial.println(dados)

Envia dados pela porta serial, seguidos automaticamente por um Enter (carriage return) e nova linha(line feed). Esta função funciona da mesma forma que o Serial.print(), mas como pula linha e retorna o cursor automaticamente, facilita a leitura em um terminal como o Serial Monitor do Arduino.

```
Serial.println(valorAnalogico);        //envia pela serial o valor da
                                       // variável 'valorAnalogico'
```

Nota: Para mais informações sobre as várias formas da função Serial.println() e da Serial.print() por favor entre no site www.Arduino.cc.
Nota do tradutor: observe que na função Serial.println() e na Serial.print() a palavra Serial começa com maiúscula e se for escrito com minúscula não vai funcionar.

O exemplo a seguir faz leitura de um pino analógico e envia esse valor pela porta serial a cada 1 segundo.

```
void setup()
{
  Serial.begin(9600);          //inicializa e configura a serial com 9600baud
}

void loop()
{
  Serial.println( analogRead(0) );          //lê o analógico 0 e envia pela serial
  delay(1000);                              //pausa por 1 segundo
}
```

Apêndice

Saída digital – Exemplo: Programa Blink

Este programa é considerado o "Hello world" da plataforma Arduino. Ele simplesmente faz um LED ligado ao pino 13 piscar uma vez por segundo, demonstrando o conceito de como ligar e desligar algo conectado ao Arduino. Na maioria das placas Arduino o pino 13 já possui um LED e um resistor ligados a esse pino.

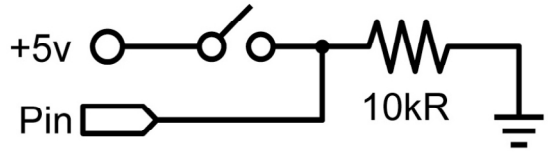
Nota do tradutor: este programa pode ser encontrado em: File>Examples>01.Basics>Blink

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

void setup()                      //roda apenas uma vez
{
  pinMode(ledPin, OUTPUT);        //configura o pino 13 como saída
}

void loop()                       //fica repetindo sem parar
{
  digitalWrite(ledPin, HIGH);     //liga o LED
  delay(1000);                    //pausa por 1 segundo
  digitalWrite(ledPin, LOW);      //desliga o LED
  delay(1000);                    //pausa por 1 segundo
}
```

Entrada Digital



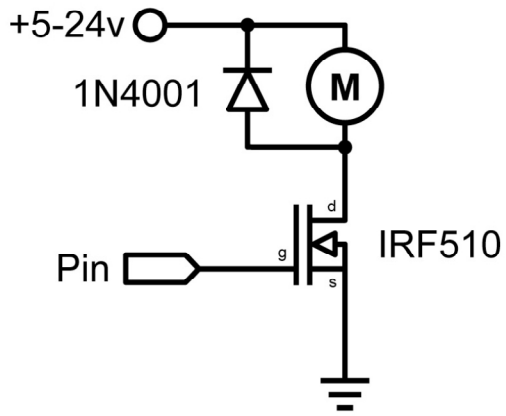
Esta é a forma mais simples de entrada com apenas dois possíveis estados: ligado ou desligado. Este exemplo lê uma chave simples ou um pushbutton conectado ao pino 2. Quando a chave é fechada a entrada será lida como HIGH e ligará em seguida um LED.

```
int ledPin = 13;        //pino de saída para conectar o led
int inPin = 2;          //pino de entrada para a chave

void setup()
{
  pinMode(ledPin, OUTPUT);      //configura pino 13 como saída
  pinMode(inPin, INPUT);        //configura pino 2 como entrada
}

void loop()
{
  if( digitalRead(inPin) == HIGH)    //a entrada está em HIGH ? (nível alto ou 5volts)
  {
    digitalWrite(ledPin, HIGH);      //liga o LED
    delay(1000);                     //pausa por 1 segundo
    digitalWrite(ledPin, LOW);       //desliga o LED
    delay(1000);                     //pausa por 1 segundo
  }
}
```

Saída de Alta Corrente



Algumas vezes é necessário controlar mais de 40 miliamperes a partir de um Arduino. Nestes casos um MOSFET ou um transistor pode resolver o caso sendo utilizado como chave para altas correntes. O exemplo a seguir liga e desliga o MOSFET 5 vezes por segundo.
Nota: o circuito acima mostra um diodo de proteção para cargas indutivas como um motor ou solenóide, para cargas não indutivas o diodo não precisa ser utilizado.

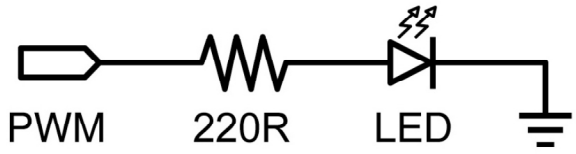
Nota do tradutor: é aconselhado instalar um dissipador de calor nos MOSFETS para evitar que eles queimem. Podem ser utilizados outros no lugar do IRF510, como o IRF540, IRFZ44, IRF3210 etc.

```
int outPin = 5;

void setup()
{
  pinMode(outPin, OUTPUT);
}

void loop()
{
  for(int i = 0; i <= 5; i++)          //repete 5 vezes
  {
  digitalWrite(outPin, HIGH);          //liga o MOSFET
  delay(250);                          //pausa ¼ de segundo
  digitalWrite(outPin, LOW);           //desliga o MOSFET
  delay(250);                          //pausa por ¼ de segundo
  }
  delay(1000);                         //pausa por 1 segundo
}
```

Saída PWM



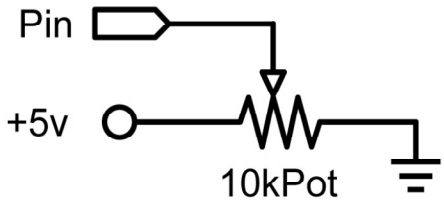
Pulse Width Modulation (PWM) significa "Modulação por Largura de Pulso" e é um falso analógico criado pulsando a saída rapidamente. Este recurso pode ser utilizado para controlar a luminosidade de um LED ou mais na frente controlar um servo motor. O exemplo a seguir lentamente aumenta e diminui o brilho de um LED conectado a um pino com PWM o efeito é obtido através de repetições for.

```
int ledPin = 9;  //pino 9 com PWM para o LED

 void setup()            //setup não foi necessário neste exemplo
{
}

void loop()                          //loop principal
{
   for(int i = 0; i<=255)            //gera os valores crescentes de i
   {
      analogWrite(ledPin, i );       //seta o brilho igual ao valor de i
      delay(100);                    //pausa por 100ms
   }
   for(int i=255; i >=0; i - -)      //gera os valores decrescentes de i
   {
      analogWrite(ledPin, i );       // seta o brilho igual ao valor de i
      delay(100);                    //pausa por 100ms
   }
}
```

Entrada Analógica – Conectando um Potenciômetro



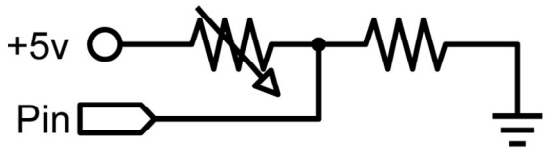
É possível utilizar um potenciômetro em uma entrada analógica do Arduino para usar seu recurso de ADC(Conversor Analógico para Digital) e obter valores de 0 a 1023. O exemplo a seguir usa um potenciômetro para controlar o a freqüência das piscadas de um LED.

```
int potPin = 0;
int ledPin = 13;

void setup()
{
   pinMode(ledPin, OUTPUT);          //declara pino 13 como saída
}

void loop()
{
   digitalWrite(ledPin, HIGH);       //liga o LED
   delay( analogRead(potPin) );      //pausa o tempo definido pelo potenciômetro
   digitalWrite(ledPin, LOW);        //desliga o LED
   delay( analogRead(potPin) );      //pausa o tempo definido pelo potenciômetro
}
```

Entrada Analógica – Conectando um resistor variável



Resistores variáveis incluem os fotoresistores, termistores, sensores de flexão(flex sensors) e por aí vai. Este exemplo faz o uso da leitura analógica e depois usa este valor para fazer um delay. Com isto a velocidade com que um LED aumenta ou diminui o brilho é modificada.

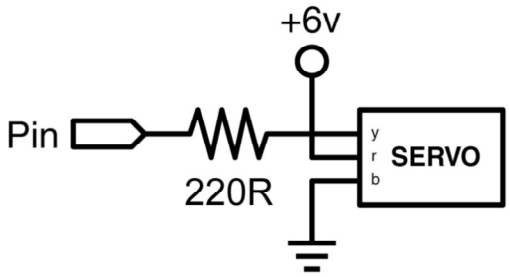
```
int ledPin = 9;          //pino PWM para o LED
int analogPin = 0;       //resistor variável no pino analógico 0

void setup()             //não foi necessário setup nesse exemplo
{}

void loop()
{
  for(int i = 0; i <= 255; i++)          //gera valores crescentes para i
  {
    analogWrite(ledPin, i);              //seta o brilho do led de acordo com i
    delay( delayVal() );                 //aguarda o tempo definido pela função delayVal()
  }                                      //retorna o valor lido pelo resistor variável
  for(int I = 255; I >= 0; I - -)        //gera valores decrescentes para i
  {
    analogWrite(ledPin, i);              //seta o brilho do LED de acordo com i
    delay( delayVal() );                 //pega o tempo da pausa e espera
  }
}

int delayVal()           //criada para ler o resistor variável e preparar o valor do tempo
{
  int v;
  v = analogRead(analogPin);             //lê o pino analógico
  v /= 8;                          //converte de 0-1023 para 0-127
  return v;                        //retorna o valor de v para quem chamou a função
}
```

Servo motor – controlando um servo motor



Servo motores de modelismo são motores que possuem um circuito de controle já acoplado a eles e que podem se mover em um arco de 180°. Tudo que é preciso é enviar um pulso a cada 20ms. Este exemplo usa a função servoPulse para mover o servo de 10 até 170 e depois retorna.

```
int servoPin = 2;        //servo conectado ao pino digital 2
int myAngle;             //armazena o ângulo atual do servo
int pulseWidth;          //variável utilizada pela função servoPulse

void setup()
{
  pinMode(servoPin, OUTPUT);              //configura pino 2 como saída
}

void servoPulse(int servoPin, int myAngle)  //função criada para gerar o pulso
{
  pulseWidth = (myAngle * 10) + 600;
  digitalWrite(servoPin, HIGH);
  delayMicroseconds(pulseWidth);
  digitalWrite(servoPin, LOW);
}

void loop()
{
  //servo inicia em 10 graus e vai rotacionando ate 170 graus
  for(myAngle = 10; myAngle <= 170; myAngle++ )
  {
    servoPulse(servoPin, myAngle); //chama função servoPulse para gerar novo pulso
    delay(20);                     //aguarda 20ms para um novo ciclo
  }
  //agora servo inicia em 170 e rotaciona para 10 graus
  for(myAngle = 170; myAngle >= 10; myAngle - -)
  {
    servoPulse(servoPin, myAngle); //chama a função para gerar o novo pulso
    delay(20);                     //aguarda 20ms para um novo ciclo
  }
}
```

Servo motor – Controlando através da <Servo.h>

Nota do tradutor: atualmente na IDE do Arduino temos dois exemplos já prontos que utilizam a biblioteca <Servo.h>. Que são o (File>Examples>Servo> Knob) que comanda um servo através da leitura de um potenciômetro na porta analógica e o outro (File>Examples>Servo> Sweep) que fica fazendo o servo ir de 0 a 180 graus e depois voltar para 0 repetidamente. Abaixo segue uma cópia traduzida do programa "Knob":

```
// Controlando a posição de um servo através de um potenciômetro
// por Michal Rinott <http://people.interaction-ivrea.it/m.rinott>

#include <Servo.h>

Servo myservo;  // cria um objeto servo com nome 'myservo'

int potpin = 0;  // pino analogico que conecta com o potenciômetro
int val;         // variavel para armazenar  o valor lido do analógico

void setup()
{
  myservo.attach(9);  // conecta o objeto servo 'myservo' ao pino 9
}

void loop()
{
  val = analogRead(potpin);          // lê o analogico (valor entre 0 e 1023)
  val = map(val, 0, 1023, 0, 179);   // muda a escala de 0-1023 para 0-179
  myservo.write(val);                //atribui o valor remapeado ao servo
  delay(15);                         //espera 15ms
}
```