Estrutura

setup()

A função *setup()* é chamada quando um esquema começa. Use-a para inicializar variáveis, modos de pino, começar a usar as bibliotecas, etc. A função de configuração será executada apenas uma vez, após cada inicialização ou reset da placa Arduino.

Exemplo:

```
int buttonPin = 3;
void setup()
{
  Serial.begin(9600);
  pinMode(buttonPin, INPUT);
}
void loop()
{
  // ...
}
```

loop()

Depois de criar uma função *setup()*, que inicializa e define os valores iniciais, a função *loop()* faz exatamente o que seu nome sugere, dá loops (ou ciclos) consecutivamente, permitindo que o seu programa mude e responda. Use-a para controlar ativamente a placa Arduino.

Exemplo

```
 const int buttonPin = 3;
// a configuração inicializa o serial e o pino do botão
void setup()
{
  Serial.begin(9600);
  pinMode(buttonPin, INPUT);
}
// o ciclo checa o pino do botão cada vez,
// e irá enviar ao serial se for pressionado
void loop()
{
  if (digitalRead(buttonPin) == HIGH)
    Serial.write('H');
  else
    Serial.write('L');

  delay(1000);
}
```

Estruturas de Controle

if (condicional) e ==, !=, <, > (operador de comparação)

if, que é usado em conjunção com um operador de comparação, testa se uma determinada condição tenha sido atingida, tal como uma entrada sobre um certo número. O formato para um *if* teste é:

```
if (someVariable > 50)
{
  // faz alguma coisa aqui
}
```

O programa testa para ver se alguma variável é maior do que 50. Se for, o programa realiza uma ação particular. Dito de outra forma, se a declaração em parênteses é verdadeira, as instruções dentro dos colchetes são executadas. Se não, o programa salta sobre o código.

Os colchetes podem ser omitidos após uma instrução if. Se isso for feito, a próxima linha (definido pelo ponto e vírgula) torna-se a única declaração condicional.

```
if (x > 120) digitalWrite(LEDpin, HIGH);

if (x > 120)
digitalWrite(LEDpin, HIGH);

if (x > 120){ digitalWrite(LEDpin, HIGH); }

if (x > 120){
  digitalWrite(LEDpin1, HIGH);
  digitalWrite(LEDpin2, HIGH);
}                              // tudo está correto
```

As declarações a serem avaliadas dentro dos parênteses exigem o uso de um ou mais operadores:

Operadores de comparação:

```
x == y (x é igual a y)
x != y (x não é igual a y)
x <  y (x é menor que y)
x >  y (x é maior que y)
x <= y (x é menor ou igual a y)
x >= y (x é maior ou igual a y)
```

Cuidado:

Cuidado com o uso acidental do sinal de igual simples (por exemplo, se (x = 10)). O único sinal de igual é o operador de atribuição, e define x a 10 (coloca o valor 10 na variável x). Em vez de usar o sinal de igual duplo (por exemplo, se (x == 10)), que é o operador de comparação, e testa se x é igual a 10 ou não. Esta última afirmação só é verdadeira se x é igual a 10, mas a declaração será sempre verdade.

Isso ocorre porque C avalia a declaração if (x = 10) da seguinte forma: 10 é atribuído a x (lembre-se que o sinal de igual simples é o operador de atribuição), então x agora contém

10. Em seguida, o "se" condicional avalia 10, o que sempre é avaliada como TRUE, uma vez que qualquer número diferente de zero é avaliada como TRUE. Consequentemente, se (x = 10) sempre irá avaliar como TRUE, o que não é o resultado desejado ao usar um *'if'*. Além disso, a variável x será definida como 10, o que também não é uma ação desejada.
if também pode ser parte de uma ramificação da estrutura de controle usando a construção if ... else.

if/else

if/else permite maior controle sobre o fluxo de código do que o básico *if*, ao permitir que múltiplos testes sejam agrupados. Por exemplo, uma entrada analógica pode ser testada e uma ação feita se a entrada for inferior a 500, e uma outra ação feita se a entrada for de 500 ou mais. O código ficaria assim:

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

else pode proceder outro if teste (teste se), então multiplicar, de modo que vários testes, mutuamente exclusivos possam ser executado ao mesmo tempo.
Cada teste prosseguirá para o próximo até que um teste verdadeiro é encontrado. Quando um teste verdadeiro for encontrado, seu bloco associado de código é executado, e em seguida, o programa salta para a linha seguinte à construção inteira if/else. Se nenhum teste provar ser verdadeiro, o mais padrão do bloco else é executado, se *if* estiver presente, e define o comportamento padrão.
Note-se que o bloco *else* pode ser usado com ou sem um bloco de terminação else e vice-versa. Um número ilimitado de ramos *else if* é permitido.

```
if (pinFiveInput < 500)
{
  // faz a coisa A
}
else if (pinFiveInput >= 1000)
{
  // faz a B
}
else
{
  // faz a C
}
```

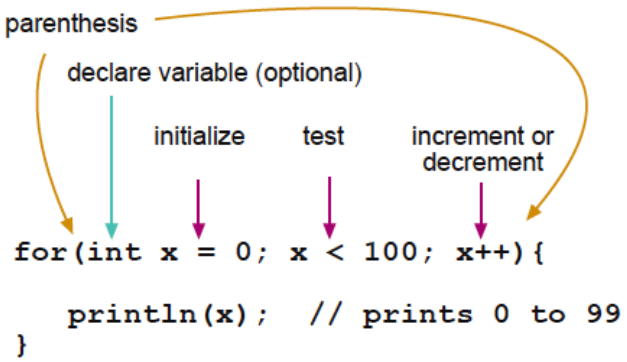
Outra maneira de expressar ramificação, testes mutuamente exclusivos, é com a instrução *switch case*.

for statements

Descrição

A instrução *for* é usado para repetir um bloco de instruções entre chaves. Um contador de incremento é geralmente usado para incrementar e finalizar o loop (ciclo). A instrução *for* é útil para qualquer operação repetitiva, e é frequentemente usada em combinação com *arrays* (sequencias) para operar em coleções de *data/pins*. Existem três partes para o cabeçalho de ciclo (loop):

```
for (initialization; condition; increment) {
//declarações;
}
```



A inicialização acontece primeiro e exatamente uma vez. Cada vez através do loop, a condição é testada e, se ele é verdadeira, o bloco de declaração, e o incremento é executado, então a condição é testada novamente. Quando a condição se torna falsa, o ciclo termina.

Exemplo

```
// Dim an LED using a PWM pin
int PWMpin = 10; // LED em série com um resistor e 470 ohm no pino 10

void setup()
{
  // nenhum configuração necessária
}

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

Coding Tips

O código C para *loop* é muito mais flexível do que loops encontrados em algumas outras linguagens de computador, incluindo BASIC. Qualquer um ou todos os três elementos de cabeçalho podem ser omitidos, embora sejam necessárias o ponto e vírgula. Também as declarações para a inicialização, condição e incremento podem ser quaisquer declarações C válidas com variáveis independentes, e utilizar quaisquer tipos de dados C, incluindo os *floats* (flutuantes). Estes tipos de declarações incomuns podem fornecer soluções para alguns problemas de programação raros.

Por exemplo, usando uma multiplicação na linha de incremento vai gerar uma progressão logarítmica:

```
for(int x = 2; x < 100; x = x * 1.5){
println(x);
}
Generates: 2,3,4,6,9,13,19,28,42,63,94
//Outro exemplo, enfraquece o LED a cada ciclo:
void loop()
{
  int x = 1;
  for (int i = 0; i > -1; i = i + x){
    analogWrite(PWMpin, i);
    if (i == 255) x = -1;            // muda a direção do pico
    delay(10);
  }
}
```

Declarações switch/case

Como declarações *if*, switch...case controla o fluxo de programas, permitindo que programadores especifiquem códigos diferentes que devem ser executadas em várias condições. Em particular, uma instrução *switch* compara o valor de uma variável para os valores especificados nas declarações de caso. Quando uma instrução case é encontrada cujo valor corresponde ao da variável, o código da declaração case é executado.

A palavra-chave *break* sai da instrução *switch*, e é normalmente usada no final de cada caso. Sem uma instrução *break*, a instrução *switch* continuará executando as seguintes expressões ("falling-through") até uma pausa, ou o fim da instrução switch ser atingida.

Exemplo

```
switch (var) {
  case 1:
    //faz alguma coisa quando var for igual a 1
    break;
  case 2:
    // faz alguma coisa quando var for igual a 2
    break;
  default:
    // Se nada mais corresponde, faz o padrão
    // Default é opcional
}
```

Sintaxe

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

Parâmetros

var: a variável cujo valor compara com os vários casos

label: um valor para comparar a variável

while loops

Descrição

O ciclo *while* irá repetir continuamente e infinitamente, até que a expressão dentro dos parênteses, () tornar-se falsa. Algo precisa mudar a variável testada, ou o ciclo *while* nunca vai sair. Isso poderia ser em seu código, como uma variável incrementada, ou uma condição externa, tais como teste de um sensor.

Sintaxe

```
while(expression){
  // declarações
}
```

Parâmetros

expression - a declaração C (boolean) que calculoa como *true* ou *false*

Exemplo

```
var = 0;
while(var < 200){
  // faz alguma coisa repetindo 200 vezes
  var++;
}
```

do - while

O ciclo *do* trabalha da mesma maneira que o ciclo *while*, com a exceção de que a condição é testada no final do ciclo, de modo que o ciclo *do* irá sempre executar pelo menos uma vez.

```
do
{
    // bloco de declarações
} while (testa a condição);
```

Exemplo

```
do
{
  delay(50);          // espera o sensor estabilizar
  x = readSensors();  // checa os sensores

} while (x < 100);
```

break

break é usado para sair do *do*, *for*, ou *loop while*, ignorando a condição de ciclo normal. Ele também é usado para sair da declaração *switch*.

Exemplo

```
for (x = 0; x < 255; x ++)
{
    digitalWrite(PWMpin, x);
    sens = analogRead(sensorPin);
    if (sens > threshold){      //salva no sensor detectado
       x = 0;
       break;
    }
    delay(50);
}
```

continue

A instrução *continue* ignora o resto da interação atual de um ciclo (*do*, *for*, ou *while*). Ela continua marcando a expressão condicional do ciclo, e prosseguindo com as interações subsequentes.

Exemplo

```
for (x = 0; x < 255; x ++)
{
    if (x > 40 && x < 120){      // cria salto nos valores
       continue;
    }

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

return

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

Sintaxe:

return;
return value; // ambas as formas são válidas

Parâmetros

value: qualquer variável ou tipo de constante

Exemplos:

Uma função para comparar uma entrada de sensor com um limiar

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

A palavra *return* é útil para testar uma seção de código sem ter que "comentar" grandes seções de código possivelmente buggy.

```
void loop(){

// idéia de código brilhante para testar aqui

return;

// resto de um esquema disfuncional aqui
// o código nunca será executado
}
```

goto

Transfere o fluxo do programa para um ponto marcado no programa

Sintaxe

label:
goto label; // envia o fluxo do programa para um registro

Dica

O uso de *goto* é desencorajado em programação C, e alguns autores de livros de programação C afirmam que o *goto* nunca é necessário, mas usado criteriosamente, ele pode simplificar certos programas. A razão pela qual muitos programadores desaprovam o uso de instruções *goto*, é fácil criar um programa com o fluxo indefinido do programa, o que nunca poderá ser depurado.
Com isso dito, há casos em que uma instrução *goto* pode vir a calhar, e simplificar a codificação. Uma dessas situações é de romper profundamente ciclos *for* aninhados, ou blocos lógicos *if*, em uma determinada condição.

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;}
            // mais declarações ...
        }
    }
}
bailout:
```

Outras Sintaxes

; ponto e vírgula

Usado no final de uma instrução.

Exemplo

```
int a = 13;
```

Dica

Esquecendo-se de colocar um ponto e vírgula no final da linha irá resultar em um erro de compilação. O texto do erro pode ser óbvio, e se refere a um ponto e vírgula faltando, ou talvez não. Se um erro do compilador impenetrável ou aparentemente ilógico surge, uma das primeiras coisas a verificar é um ponto e vírgula faltando, nas imediações, que precede a linha em que o compilador reclamou.

{} chaves

Chaves (também conhecidas como "chaves" ou como "entre chaves") são uma parte importante da linguagem de programação C. Elas são usadas em várias construções diferentes, descritas a seguir, e isso às vezes pode ser confuso para os iniciantes.

Uma chave de abertura "{" deve ser sempre seguido por uma chave de fechamento "}". Esta é uma condição que é muitas vezes referida como o código sendo equilibrado. A IDE do Arduino (ambiente de desenvolvimento integrado) inclui um recurso conveniente para verificar o saldo de chaves. Basta selecionar uma chave, ou mesmo clique no ponto de inserção imediatamente após uma chave, e seu companheiro lógico será destacado.

Atualmente, este recurso é um pouco buggy como o IDE vai encontrar muitas vezes (incorretamente) uma chave no texto que foi "como comentário".

Programadores iniciantes e programadores que vêm do C a partir da linguagem BASIC encontram-se muitas vezes usando chaves confusas ou difícil. Afinal de contas, as mesmas chaves substituem a instrução *RETURN* em uma sub-rotina (função), a declaração *ENDIF* em uma condicional e a instrução Next em um ciclo *FOR*.

Porque o uso da chave é tão variada, é uma boa prática de programação digitar a chave de fechamento imediatamente após digitar a chave de abertura ao inserir uma construção que requer chaves. Em seguida, inserir algumas linhas entre suas chaves no começo da inserção das declarações. Suas chaves, e sua atitude, nunca se tornará desequilibrada.

Chaves desequilibradas muitas vezes podem levar a erros de compilação, que às vezes podem ser de difícil rastreamento em um grande programa. Por causa de seus variados usos, as chaves também são extremamente importantes para a sintaxe de um programa e mover uma chave de uma ou duas linhas, muitas vezes afeta drasticamente o significado de um programa.

O principal uso de chaves

Funções

```
void myfunction(datatype argument){
  declarações
}
```

Ciclos

```
while (boolean expression)
{
  declarações
}

do
{
  declarações
} while (boolean expression);

for (initialisation; termination condition; incrementing expr)
{
  declarações
}
```

Instruções condicionais

```
if (boolean expression)
{
  declarações
}

else if (boolean expression)
{
  declarações
}
else
{
  declarações
}
```

Comments (Comentários)

Os comentários são linhas do programa que são usados para informar a si mesmo ou a outros sobre a forma como o programa funciona. Eles são ignorados pelo compilador, e não exportados para o processador, para que

eles não ocupem qualquer espaço no chip Atmega.
Comente apenas propósitos que irão ajudar você a entender (ou lembrar) como o programa funciona ou para informar aos outros como seu programa funciona. Há duas maneiras diferentes de marcação de uma linha como um comentário:

Exemplo

```
 x = 5;  // Isto é uma linha de comentário. Tudo após os traços são comentários
        // para o final da linha

/* é um comentário de múltiplas linhas – usado para comentar blocos de código

if (gwb == 0){    // única linha de comentário é OK dentro de um comentário de várias linhas
x = 3;            /* mas não em outro comentário de várias linhas - isto é inválido */
}
// não se esqueça do "fechamento" comentário - eles têm que ser equilibrado!
*/
```

Dica

Ao experimentar com o código "comentando" partes do seu programa é uma forma conveniente para remover as linhas que podem ser bug. Isso deixa as linhas do código, mas transforma-os em comentários, para que o compilador simplesmente ignore-as. Isso pode ser especialmente útil quando se tenta localizar um problema, ou quando um programa se recusa a compilar e o erro do compilador é crítico ou inútil.

#define

#define é um componente C útil que permite o programador dar um nome a um valor constante antes que o programa seja compilado. Constantes definidas no arduino não ocupam qualquer espaço de memória do programa no chip. O compilador irá substituir as referências a essas constantes com o valor definido no tempo da compilação.
Isto pode ter alguns efeitos colaterais indesejados, se por exemplo, um nome constante que tinha sido #definido está incluído em qualquer outro nome constante ou variável. Nesse caso, o texto será substituído pelo número #definido (ou texto).
Em geral, a palavra-chave *const* é preferido para a definição de constantes e deve ser utilizado em vez do *#define*.

O define no Arduino têm a mesma sintaxe do define no C:

Sintaxe

```
#define constantName value
```

Note que o # é necessário.

Exemplo

```
#define ledPin 3
// O compilador irá substituir qualquer menção ao ledPin com o valor 3 na hora da compilação.
```

Dica

Não existe nenhum ";" depois da instrução *#define*. Se você incluir uma, o compilador irá lançar erros críticos na parte inferior da página.
#define ledPin 3; // isto é um erro

Similarmente, incluir um sinal de igual depois da instrução *#define* irá gerar um erro crítico de compilação no final da página.
#define ledPin = 3 // isto também é um erro

#include

#include é usado para incluir bibliotecas externas em seu esboço. Isto dá ao programador acesso a um grande grupo de bibliotecas padrão C (grupos de funções pré-fabricados), e também bibliotecas escritas especialmente para Arduino.
A página de referência principal para as bibliotecas AVR C (AVR é uma referência para os chips Atmel em que o Arduino é baseado) está aqui.
Note que *#include*, semelhante ao *#define*, não tem nenhum ponto e vírgula no final da linha, e o compilador irá produzir mensagens de erro críticos se você adicionar um.

Exemplo

Este exemplo inclui uma biblioteca que é usada para colocar os dados no programa da memória flash em vez da RAM. Isso economiza o espaço de memória RAM para as necessidades de memória dinâmica e torna práticas grandes tabelas de pesquisa.

```
# 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};
```

Operadores Aritméticos

= operador de atribuição (sinal de igual simples)

Armazena o valor à direita do sinal de igualdade na variável à esquerda do sinal de igualdade.
O sinal de igual simples na linguagem de programação C é chamado de operador de atribuição. Ela tem um significado diferente do que na aula de álgebra, onde indica uma equação ou igualdade. O operador de atribuição informa o microcontrolador para avaliar qualquer valor ou expressão no lado direito do sinal de igualdade e armazená-lo na variável à esquerda do sinal de igual.

Exemplo

```
int sensVal;                    // declara um inteiro variável nomeado sensVal
senVal = analogRead(0);         // armazena a voltagem de entrada (digitalizada) no pino analógico 0 no SensVal
```

Dicas de programação

A variável no lado esquerdo do operador de atribuição (sinal =) deve ser capaz de manter o valor armazenado na mesma. Se não for grande o suficiente para conter um valor, o valor armazenado na variável será incorreta.
Não confunda o operador de atribuição de [=] (sinal de igual simples) com o operador de comparação [==] (sinal de igual duplo), que avalia se duas expressões são iguais.

_if_ (condicional) e ==,!=, <, > (operadores de comparação)

_if_, que é usado em conjunção com um operador de comparação, testa se um determinado estado foi atingido, tal como uma entrada de um certo número. O formato para um caso de teste _if_ é:

```
if (someVariable > 50)
{
  // faz alguma coisa aqui
}
```

O programa testa para ver se alguma variavel é maior do que 50. Se for, o programa realiza uma ação particular. Dito de outra forma, se a declaração em parênteses é verdadeira, as instruções dentro dos colchetes serão executadas. Se não, o programa salta sobre o código.
Os colchetes podem ser omitidos após uma instrução _if_. Se isso for feito, a próxima linha (definida pelo ponto e vírgula) torna-se a única declaração condicional.

```
if (x > 120) digitalWrite(LEDpin, HIGH);

if (x > 120)
digitalWrite(LEDpin, HIGH);

if (x > 120){ digitalWrite(LEDpin, HIGH); }

if (x > 120){
  digitalWrite(LEDpin1, HIGH);
  digitalWrite(LEDpin2, HIGH);
}                               // tudo está correto
```

As declarações sendo avaliados dentro dos parênteses requerem o uso de um ou mais operadores:

Operadores de Comparação:

x == y (x é igual a y)
x != y (x não é igual a y)
x < y (x é menor que y)
x > y (x é maior que y)
x <= y (x é menor ou igual a y)
x >= y (x é maior ou igual a y)

Cuidado:

Cuidado com o uso acidental do sinal de igual simples (por exemplo, se (x = 10)). O único sinal de igual é o operador de atribuição, e define x a 10 (coloca o valor 10 para a variável x). Em vez de usar o sinal de igual duplo (por exemplo, se (x == 10)), que é o operador de comparação, e testa se x é igual a 10 ou não. Esta última afirmação só é verdadeira se x for igual a 10, mas a primeira declaração será sempre verdadeira.
Isso ocorre porque C avalia a declaração _if_ (x = 10) da seguinte forma: 10 é atribuído a x (lembre-se que o sinal de igual simples é o operador de atribuição), então x agora contém 10. Em seguida, o "se" condicional avalia 10, o que sempre é avaliada como verdadeiro, uma vez que qualquer número diferente de zero é avaliada como verdadeiro. Conseqüentemente, se (x = 10) sempre irá avaliar como verdadeiro, o que não é o resultado desejado ao usar uma declaração _'if'_. Além disso, a variável x será definida como 10, o que também não é uma ação desejada.
_if_ também pode ser parte de uma estrutura de controle de ramificação usando a construção if ... else.

Operadores Booleanos

Operadores Booleanos
Estes podem ser usados dentro da condição de uma declaração _if_.
&& (e lógico)
Verdadeiro somente se ambos os operandos são verdadeiros, por exemplo:

```
if (digitalRead(2) == HIGH  && digitalRead(3) == HIGH) {
// lê dois interruptores
  // ...
}
```
É verdadeiro apenas se ambas as entradas forem HIGH.

|| (lógico ou)

True se ambos operadores forem verdadeiros, exemplo.
```
if (x > 0 || y > 0) {
  // ...
}
```
É true se ambos x e y forem maiores que 0.

! (não)

True se o operador for falso, exemplo
```
if (!x) {
  // ...
}
```

É verdadeiro se x for falso (exemplo, se x igual a 0).

Cuidado

Certifique-se de não confundir o operador boleano binário AND, && (e comercial duplo) com o operador AND & (e comercial simples). Eles são completamente diferentes.
Da mesma forma, não confunda o || booleano (duplo tubo) com o operador binário OR operador | (único tubo).
O operador binário não ~ (til) parece muito diferente do que o booleano não! (ponto de exclamação ou "ímpeto" como os programadores dizem), mas você ainda precisa ter certeza de onde você quer eles.

Exemplos

```
if (a >= 10 && a <= 20){}    // verdadeiro se estiver entre 10 e 20
```

Operadores Ponteiro de Acesso

Os operadores de ponteiro
& (referência) e * (desreferência)

Os ponteiros são um dos temas mais complicados para os iniciantes no aprendizado de C, e é possível escrever a grande maioria dos esquemas Arduino sem nunca encontrar ponteiros. No entanto, para a manipulação de determinadas estruturas de dados, o uso de ponteiros pode simplificar o código e, conhecimento de manipulação de ponteiros é útil para ter em sua caixa de ferramentas.

Operadores Binários (Bitwise)

Bitwise AND (&), Bitwise OR (|), Bitwise XOR (^)

Binário AND (&)

Os operadores binários executam seus cálculos ao nível de variáveis bit. Eles ajudam a resolver uma ampla gama de problemas de programação comuns. Grande parte do material abaixo é de um excelente tutorial sobre binários de matemática que podem ser encontrada aqui.

Descrição e Sintaxe

Abaixo estão as descrições e Sintaxe para todos os operadores. Mais detalhes podem ser encontrados no tutorial referenciado.

Binário AND (&)

O operador binário AND em C++ é um único E comercial, &, usado entre duas outras expressões inteiras. Binário AND opera em cada posição de bit nas expressões circundantes independentemente, de acordo com a seguinte regra: se os dois bits de entrada são 1, a saída resultante é um, de outro modo a saída é 0. Outra forma de expressar é:

```
  0  0  1  1    operador 1
  0  1  0  1    operador 2
  ----------
  0   0   0   1     (operador 1 e operador 2) resultado retornado
```

No Arduino, o tipo int é um valor de 16 bits, portanto, usando & entre duas expressões int faz com que 16 operações & simultâneas ocorram. Em um fragmento de código como:

```
  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 de a e b são processadas usando o binário AND, e todos os 16 bits resultantes são armazenados na c, resultando no valor 01000100 em binário, que é de 68 em decimal.
Um dos usos mais comuns do binário AND é selecionar um bit particular (ou bits) a partir de um valor inteiro, muitas vezes chamado de mascaramento.

Binário OR (|)

O operador binário OR em C++ é o símbolo de barra vertical, |. Como o operador &, | opera de forma independente cada bit ao redor de suas

duas expressões inteiras, mas o que ele faz é diferente (é claro). O binário OR de dois bits é 1 se um ou ambos os bits de entrada forem 1, caso contrário, é 0. Por outras palavras:

```
 0 0 1 1   operador 1
 0 1 0 1   operador 2
 ----------
 0 1 1 1    (operador 1 / operador 2) – resultado retornado
```

Aqui um exemplo do binário Or usado em um fragmento de código C++:

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

Exemplo de Programa

Um trabalho comum para os operadores binários AND e OR é o que os programadores chamam Leia-Modique-Escreva em uma porta. Em microcontroladores, uma porta é um número de 8 bit que representa algo sobre a condição dos pinos. Escrevendo para uma porta controla todos os pinos de uma só vez.
PORTD é uma constante interna que se refere aos estados dos pinos digitais de saída 0,1,2,3,4,5,6,7. Se houver 1 em uma posição bit, então que o pino é *HIGH*. (Os pinos já precisam ser definidos para *OUTPUT* com o comando *pinMode()*). Então, se nós escrevermos PORTD = B00110001; fizermos pinos 2,3 e 7 *HIGH*. Um pequeno problema aqui é que também pode ter alterado o estado de pinos 0 e 1, que são usados pelo Arduino para comunicação serial para que possamos ter interferido com a comunicação serial.

Nosso algoritmo do programa é:

• Obter PORTD e limpar somente os bits correspondentes aos pinos que deseja controlar (com binário AND).
• Combinar o valor PORTD modificado com o novo valor para os pinos sob controle (com binário OR).

```
int i;    // contador variável
int j;

void setup(){
DDRD = DDRD | B11111100; // declara a direção do bos bits para do pino de bits 2 para 7, deixa 0 e 1 intocados (xx | 00 == xx)
// igual ao pinMode(pin, OUTPUT) para os pinos 2 a 7
Serial.begin(9600);
}

void loop(){
for (i=0; i<64; i++){

PORTD = PORTD & B00000011;  // clear out bits 2 - 7, leave pins 0 and 1 untouched (xx & 11 == xx)
j = (i << 2);               // muda a variável dos pinos 2 - 7 – para evitar os pinos 0 e 1
PORTD = PORTD | j;          // combina as informações da porta com as novas informações para os pinos de LED
Serial.println(PORTD, BIN); // depura para mostrar mascaramento
delay(100);
  }
}
```

Binário XOR (^)

Há um operador de um tanto incomum em C++ chamado binário EXCLUSIVE OR, também conhecido como binário XOR. (Em Inglês isso geralmente é pronunciado "EKS-or".) O operador binário XOR é escrito usando o símbolo de acento circunflexo ^. Este operador é muito semelhante ao do operador binário OR |, apenas avalia a 0 para uma determinada posição de bit quando ambos os bits de entrada para essa posição são 1:

```
 0 0 1 1   operador 1
 0 1 0 1   operador 2
 ----------
 0 1 1 0    (operador 1 ^ operador 2) – resultado retornado
```

Outra maneira de olhar para XOR é que cada bit no resultado é 1 se os bits de entrada são diferentes, ou 0 se eles são os mesmos.
Aqui está um exemplo de código simples:

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

O operador ^ é muitas vezes usado para alternar (ou seja, mudança de 0 para 1 ou 1 para 0) alguns dos bits em uma expressão inteira. Em uma operação binária OR, se houver um 1 no bit de máscara, que é um bit invertido, se existir um 0, o bit não é invertido e permanece o mesmo. Abaixo está um programa para piscar no pino digital 5.

```
// Blink_Pin_5
// demo para OR Exclusivo
void setup(){
DDRD = DDRD | B00100000; // define o pino digital 5 como OUTPUT
Serial.begin(9600);
}

void loop(){
```

```
PORTD = PORTD ^ B00100000;  // inverte bit 5 (pino digital 5), deixa os outros intocados
delay(100);
}
```

Operadores Compostos

++ (incremento) / -- (decremento)

Descrição

Incrementa ou decrementa (aumenta ou diminui) a variável

Sintaxe

```
x + +; / / incrementa x por um e retorna o valor antigo de x
+ + x / / incrementa x por um e retorna o novo valor de x

x - / / decrementa x por um e retorna o valor antigo de x
- x / / decrementa x por um e retorna o novo valor de x
```

Parâmetros

x: um inteiro (integer) ou longo (long) - possivelmente não assinado

Retorna

O valor original ou recém-incrementado/decrementado da variável.

Exemplos

```
x = 2;
y = + + x / / x agora contém 3, e contém 3
y = x - / / x contém 2 novamente, e ainda contém 3
```

+= , -= , *= , /=

Descrição

Executa uma operação matemática em uma variável com outra constante ou variável. Os operadores + = (et al) são apenas uma abreviação conveniente para a sintaxe expandida, listados abaixo.

Sintaxe

```
x += y;   // equivalente a expressão x = x + y;
x -= y;   // equivalente a expressão x = x - y;
x *= y;   // equivalente a expressão x = x * y;
x /= y;   // equivalente a expressão x = x / y;
```

Parâmetros

x: qualquer tipo de variável
y: qualquer tipo de variável ou constante

Exemplos

```
x = 2;
x += 4;      // x agora contém 6
x -= 3;      // x agora contém 3
x *= 10;     // x agora contém 30
x /= 2;      // x agora contém 15
```

lógica binária composta AND (&=)

Descrição

O operador binário (bitwise) composto AND (& =) é muitas vezes usado com uma variável e uma constante para forçar os bits específicos em uma variável para o estado *LOW* (a 0). Esta é muitas vezes referida em guias de programação como "limpeza" ou "reiniciação" dos bits.

Sintaxe:

```
x &= y;   // equivalente a x = x & y;
```

Parâmetros

x: um char, int ou variável long
y: uma constante integer ou char, int ou long

Exemplo:

Primeiro, a revisão do operador binário AND (&)

```
  0  0  1  1    operador 1
  0  1  0  1    operador 2
  ----------
  0  0  0  1     (operador 1 & operador 2) – resultado retornado
```

Bits que são "binários ANDed" com 0 a 0 são apagadas assim, se myByte é uma variável byte,
myByte & B00000000 = 0;
Bits que são "binários ANDed" com 1 mantêm-se inalterados por isso,
myByte & B11111111 = myByte;

Nota: porque estamos a lidar com bits em um operador binário - é conveniente usar o formatador binário com constantes. Os números ainda são o mesmo valor em outras representações, eles não são tão fáceis de entender. Além disso, B00000000 é mostrado para maior clareza, mas o zero em qualquer formato de número é zero (hmmm algo filosófico lá?)
Consequentemente - para limpar (colocar zero) os bits 0 e 1 de uma variável, deixando o resto da variável inalterada, use o Operador binário Composto AND (& =) com uma Constante B11111100

```
  1  0  1  0  1  0  1  0   variaável
  1  1  1  1  1  1  0  0   máscara
  ----------------------
  1  0  1  0  1  0  0  0

 Variável não modificada
              bits apagados
```

Aqui a mesma representação com a variável bits substituída pelo símbolo x

```
x x x x x x x x   variable
1 1 1 1 1 1 0 0   mask
----------------------
x x x x x x 0 0
```

Variável não modificada
bits apagados

Então:
myByte = 10101010;

myByte &= B1111100 == B10101000;

Variáveis

Constantes

Constantes
Constantes são variáveis pré-definidas na linguagem do Arduino. Elas são usadas para tornar os programas mais fáceis de ler. Classificamos constantes em grupos.
Definindo níveis lógicos, verdadeiros e falsos (booleanos Constantes)
Há duas constantes usadas para representar a verdade e a falsidade na linguagem Arduino: *true* e *false*.
false
false é a mais fácil dos dois para definir. false é definido como 0 (zero).
true
True é frequentemente dito ser definido como 1, o que é correto, mas *True* tem uma definição mais ampla. Qualquer inteiro que é diferente de zero é verdade no sentido booleano. Então, -1, 2 e -200 são todos definidos como *true*, também, em um sentido booleano.
Note-se que as constantes *true* e *false* são digitadas em letras minúsculas ao contrário de HIGH, LOW, INPUT e OUTPUT.
Definindo níveis de Pin, HIGH e LOW
Ao ler ou escrever em um pino digital há apenas dois valores possíveis que um pino pode tomar/determinado: *HIGH* e *LOW*.
HIGH
O significado de *HIGH* (em referência a um pino) é um pouco diferente dependendo se um pino é configurado como *INPUT* ou *OUTPUT*. Quando um pino é configurado como uma entrada com *pinMode*, e lido com *digitalRead*, o microcontrolador irá reportar *HIGH* se uma tensão de 3 volts ou mais está presente no pino.
Um pino também pode ser configurado como uma *INPUT* com *pinMode*, e, posteriormente, definido como *HIGH* com *digitalWrite*, isso vai definir os resistores pull-up de 20k internos, que irão orientar o pino de entrada para uma leitura *HIGH*, a menos que ele seja puxado para *LOW* pelo circuito externo. É assim que funciona tão bem o *INPUT_PULLUP*.
Quando um pino é configurado para *OUTPUT* com *pinMode*, e marcado como HIGH com digitalWrite, o pino é de 5 volts. Neste estado, pode ser uma fonte de corrente, por exemplo, acender um LED que está conectado através de um resistor em série para o ground ou para o outro pino configurado como uma saída, e definida para *LOW*.
LOW
O significado de *LOW* também tem um significado diferente, dependendo se um pino é configurado como *INPUT* ou *OUTPUT*. Quando um pino é configurado como uma entrada com *pinMode*, e lido com *digitalRead*, o microcontrolador irá reportar *LOW* se uma tensão de 2 volts ou menos está presente no pino.
Quando um pino é configurado para *OUTPUT* com *pinMode*, e marcado como *LOW* com *digitalWrite*, o pino é de 0 volts. Nesse estado ele pode dissipar a corrente, por exemplo, acender um LED que está conectado através de um resistor em série para, +5 volts, ou a outro pino configurado como uma saída, e definido como *HIGH*.
Definindo os pinos digitais, INPUT, INPUT_PULLUP, and OUTPUT
Pinos digitais podem ser usados como *INPUT*, *INPUT_PULLUP* ou *OUTPUT*. Mudando o pino com *pinMode()* muda o comportamento elétrico do pino.
Pinos configurados como INPUT
Pinos Arduino(Atmega) configurados como *INPUT* (entrada) com *pinMode()* estão a ser dito em um estado de alta impedância. Pinos configurados como *INPUT* tornam extremamente pequenas as exigências sobre o circuito de amostragem, que são, o que equivale a uma resistência em série de 100 Megohms em frente do pino. Isto torna-os úteis para a leitura de um sensor, mas não para alimentar um LED.
Se você tiver seu pino configurado como *INPUT*, você vai querer o pino tenha uma referência ao terra, muitas vezes realizado com um resistor *pull-down* (um resistor que vai ao

terra), como descrito no tutorial de serial de leitura digital.

Pinos configurados como INPUT_PULLUP

O chip Atmega do Arduino tem resistores pull-up internos (resistores que ligam a fonte internamente) que você pode acessar. Se você preferir usar estes em vez de resistores de pull-down externos, você pode usar o argumento *INPUT_PULLUP* em *pinMode().* Isso inverte efetivamente o comportamento, onde *HIGH* significa que o sensor está desligado, e *LOW* significa que o sensor está ligado. Veja o tutorial entrada serial Pullup para um Exemplo disso em uso.

Pinos configurados como Outputs

Pinos configurados como *OUTPUT* (saída) com *pinMode()* são ditos em um estado de baixa impedância. Isto significa que eles podem fornecer uma quantidade substancial de corrente para outros circuitos. Pinos Atmega podem ser fonte (fornecer corrente positiva) ou diminuir (fornecer corrente negativa) até 40 mA (miliamperes) de corrente para outros dispositivos/circuitos. Isso faz com que eles sejam úteis para alimentar LEDs, mas inútil para sensores de leitura. Pinos configurados como saídas também podem ser danificados ou destruídos, se curto-circuito entre o trilho de alimentação de 5 volts ao terra. 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.

Constantes Inteiras

As constantes inteiras são números usados diretamente em um esboço, como 123. Por padrão, esses números são tratados como Int, mas você pode mudar isso com os modificadores de U e L (veja abaixo).

Normalmente, constantes inteiras são tratadas como inteiros de base 10 (decimal), mas notação especial (formatadores) podem ser usados para digitar números em outras bases.

Base Exemplo Formato Comentário

10 (decimal) 123 nenhum

2 (binário) B1111011 comando 'B' apenas trabalha com valores de 8-bit (0 a 255) caracteres 0-1 valid

8 (octal) 0173 comando "0" caracteres 0-7 válido

16 (hexadecimal) 0x7B comando "0x" caracteres 0-9, A-F, a-f válido

Decimal tem base 10. Esta é a matemática do senso comum com o qual você está familiarizado. Constantes sem outros prefixos estão a ser assumida no formato decimal.

Exemplo:

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

Binário é base dois. Somente os caracteres 0 e 1 são válidos.

Exemplo:

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

O formatador binário só funciona em bytes (8 bits) entre 0 (B0) e 255 (B11111111). Se for conveniente, a entrada de um int (16 bits) em forma binária é você pode faze-lo em um procedimento de dois passos, tais como:

```
myInt = (B11001100 * 256) + B10101010;   // B11001100 is the high byte
```

Octal é base oito. Somente caracteres de 0 a 7 são válidos. Valores octal são indicados pelo prefixo "0"

Exemplo:

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

Aviso

É possível gerar um erro difícil de ser encontrardo (involuntariamente), incluindo um zero à esquerda antes de uma constante e ter o compilador involuntariamente interpretar seu constante como octal.

Hexadecimal (ou hex) é de base dezesseis. Os caracteres válidos são de 0 a 9 e letras de A a F; A tem o valor 10, B é 11, até F, que é 15. Valores Hex são indicadas pelo prefixo "0x". Note-se que a A-F pode ser sintetizado em maiúsculas ou minúsculas (a-f).

Exemplo:

```
0x101    // o mesmo que 257 decimal    ((1 * 16^2) + (0 * 16^1) + 1)
```

Formatadores U e L

Por padrão, uma constante inteira é tratada como um int com as limitações inerentes em valores. Para especificar uma constante inteira com outro tipo de dados, segui-a com:

• um 'u' ou 'U' para forçar a constante em um formato de dados não assinado.
Exemplo: 33U

• um 'l' ou 'L' para forçar a constante em um formato de dados de comprimento.
Exemplo: 100000L

• um 'ul' ou 'UL' para forçar a constante em uma constante longa sem sinal.
Exemplo: 32767ul

constantes de ponto flutuante

Similar para constantes inteiras, as constantes de ponto flutuante são usados para tornar o código mais legível. Constantes de ponto flutuante são trocadas durante a compilação para o valor para o qual a expressão é avaliada.

Exemplos:

n = 005;
Constantes de ponto flutuante pode também ser expressa numa variedade de notação científica. 'E' e 'e' Aceitam-se os indicadores expoente válidos.

ponto flutuante avaliada como: também avalia a:

constante		
10.0	10	
2.34E5	2.34 * 10^5	234000
67e-12	67.0 * 10^-12	.000000000067

Tipos de Dados

void

A palavra-chave *void* é usado apenas em declarações de função. Ele indica que a função não deve retornar nenhuma informação para a função a partir da qual ela foi chamada.

Exemplo:

```
/ / Ações são realizadas na função "setup" e "loop"
/ / Mas nenhuma informação é relatada para programas maiores

void setup()
{
  // ...
}

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

boolean

Um booleano detém um dos dois valores, *true* ou *false*. (Cada variável booleana ocupa um byte de memória).

Exemplo

```
int LEDpin = 5;       // LED no pino 5
int switchPin = 13;    // Interruptor momentâneo em 13, outro lado ligado ao terra
boolean running = false;

void setup()
{
  pinMode(LEDpin, OUTPUT);
  pinMode(switchPin, INPUT);
  digitalWrite(switchPin, HIGH);          // ligado o resistor pullup
}

void loop()
{
  if (digitalRead(switchPin) == LOW)
  {  // interruptor é pressionado - pullup mantém pin HIGH normalmente
    delay(100);                       // atrasa a estabilização do interruptor
    running = !running;                    // alternar a execução variável
    digitalWrite(LEDpin, running)      // indica via LED
  }
}
```

char

Descrição

Um tipo de dados que ocupa 1 byte de memória que armazena um caractere. Caracteres literais são escritos entre aspas simples, como este: 'A' (para vários caracteres - sequencias - use aspas: "ABC").

Os caracteres são armazenados como números. No entanto você pode ver a codificação específica na tabela ASCII. Isto significa que é possível fazer aritmética em caracteres, em que o valor do carácter ASCII é utilizado (por exemplo, 'A' + 1 tem o valor de 66, dado que o valor ASCII da letra maiúscula A é de 65). Ver referência *Serial.println* para saber mais sobre como os caracteres são convertidos para números.

O tipo de dados *char* é um tipo assinado, o que significa que ele codifica os números de -128 a 127. Para um tipo de dados um-byte (8 bits) não assinado, use o tipo de dados byte.

Exemplo

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

unsigned char

Descrição

Um tipo de dados sem sinal (*unsigned*) ocupa 1 byte de memória. O mesmo que o tipo de dados byte.
O tipo de dados *char* sem sinal (*unsigned char*) codifica os números de 0 a 255.
Para manter a consistência de estilo da programação Arduino, o tipo de dados byte deve ser preferido.

Exemplo

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

char()

Descrição

Converte o valor para o tipo de dados *char*.

Sintaxe

char(x)

Parâmetros

x: o valor de qualquer tipo

Respostas

char

byte

Descrição

Um byte armazena um número sem sinal de 8 bits, de 0 a 255.

Exemplo

```
byte b = B10010;  // "B" está no formato binary (B10010 = 18 decimal)
```

int

Descrição

Inteiros (*int*) são o tipo principal de dados para o armazenamento de número.
No Arduino Uno (e outras placas baseadas no ATmega) um inteiro armazena um valor de 16 bits (2 bytes). Isto origina uma gama de 32.768 a 32.767 (valor mínimo de 2 ^ 15, e um valor máximo de (2 ^ 15) - 1).
Para o Arduino Due, um inteiro armazena um valor de 32 bits (4 bytes). Isto origina uma gama de -2,147,483,648 a 2.147.483.647 (valor mínimo de 2 ^ 31 e um valor máximo de (2 ^ 31 - 1).
Inteiros armazenam números negativos com uma técnica chamada de complemento de 2 de matemática. O bit mais alto, por vezes referido como o “sinal” bit, sinalizadores do número são como número negativo. O resto dos bits são invertidos e adiciona-se 1.
O Arduino cuida de lidar com números negativos para você, para que as operações aritméticas trabalhem de forma transparente e da maneira esperada. Não pode haver uma complicação inesperada em lidar com o operador direito BitShift (>>), no entanto.

Exemplo

```
int ledPin = 13;
```

Sintaxe

```
int var = val;
```

- var – o nome da sua variável int
- val – o valor que você atribui a variável

Dica de codificação

Quando as variáveis são feitas para superar sua capacidade máxima elas "invertem" e voltam para a sua capacidade mínima, note que isso acontece nos dois sentidos. Exemplo para um inteiro de 16 bits:

```
int x;
x = -32768;
x = x - 1;      // x agora contém 32,767 – vira para direção negativa

x = 32767;
x = x + 1;      // x agora contém -32,768 - vira
```

unsigned int

Descrição

No Uno e outras placas baseadas ATMEGA, inteiros sem sinal (*unsigned ints*) são os mesmos que inteiros em que eles armazenam um valor de byte 2. Em vez de armazenar os números negativos no entanto eles só armazenam valores positivos, dando origem a uma gama útil de 0 a 65.535 (2 ^ 16) - 1).
O Due armazena um valor de 4 bytes (32 bits), que varia de 0 a 4.294.967.295 (2 ^ 32 - 1).
A diferença entre inteiros sem sinal e inteiros, está na maneira como o bit mais alto, às vezes referido como o bit de "sinal", é interpretado. No Arduino o tipo inteiro (que tem sinal), se o bit é um "1", o número é interpretado como um número negativo, e os outros 15 bits são interpretados como complemento de 2 de matemática.

Exemplo

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

Sintaxe

```
unsigned int var = val;
```

- var – o nome da sua *unsigned int*
- val – o valor declarado para a variável

Dica de codificação

Quando as variáveis são feitas para superar sua capacidade máxima elas "invertem" e voltam a sua capacidade mínima, note que isso acontece em ambos os sentidos

```
unsigned int x
x = 0;
```

```
x = x - 1;        // x agora contém 65535 – vira para a direção negativa
x = x + 1;        // x agora contém 0 - vira
```

Constantes Inteiras

As constantes inteiras são números usados diretamente em um esquema, como 123. Por padrão, esses números são tratados como inteiros, mas você pode mudar isso com os modificadores U e L (veja abaixo).
Normalmente, constantes inteiras são tratadas como inteiros de base 10 (decimal), mas notação especial (formatadores) podem ser usadas para digitar números em outras bases.

Base	Example	Formatter	Comment
10 (decimal)	123	none	
2 (binary)	B1111011	leading 'B'	only works with 8 bit values (0 to 255)
			characters 0-1 valid
8 (octal)	0173	leading "0"	characters 0-7 valid
16 (hexadecimal)	0x7B	leading "0x"	characters 0-9, A-F, a-f valid

Decimal é base 10. Esta é a matemática do senso comum, com o qual você está familiarizado. Constantes sem outros prefixos estão a ser assumida no formato decimal.
Exemplo:

```
101    // same as 101 decimal   ((1 * 10^2) + (0 * 10^1) + 1)
```

Binário é base dois. Somente caracteres 0 e 1 são válidos.
Exemplo:

```
B101    // same as 5 decimal   ((1 * 2^2) + (0 * 2^1) + 1)
```

O formatador binário só funciona em bytes (8 bits) entre 0 (B0) e 255 (B11111111). Se for conveniente, é possível a entrada de um inteiro (16 bits) em forma binária, você pode fazer isto em um procedimento de dois passos, tais como:

```
myInt = (B11001100 * 256) + B10101010;    // B11001100 is the high byte
```

Octal é a base oito. Somente caracteres de 0 a 7 são válidos. Valores octal são indicado pelo prefixo "0"
Exemplo:

```
0101    // same as 65 decimal   ((1 * 8^2) + (0 * 8^1) + 1)
```

Aviso
É possível gerar um erro difícil de ser encontrado (involuntariamente), incluindo um zero à esquerda antes de uma constante e fazer o compilador involuntariamente interpretar sua constante como octal.
Hexadecimal (ou hex) é base dezesseis. Os caracteres válidos são de 0 a 9 e as letras de A a F; A tem o valor 10, B é 11, até F, que é 15. Valores Hex são indicadas pelo prefixo "0x". Note-se que a A-F pode ser obsevado em maiúsculas ou minúsculas (af).
Exemplo:

```
0x101    // same as 257 decimal   ((1 * 16^2) + (0 * 16^1) + 1)
```

Formatadores U e L

Por padrão, uma constante inteira é tratada como um inteiro com as limitações inerentes em valores. Para especificar uma constante inteira com outro tipo de dados, segui-o com:
• um 'u' ou 'U' para forçar a constante ao formato de dados sem sinal.
Exemplo:

```
33U
```

• um 'l' ou 'L' para forçar a constante ao formato de dados de comprimento.
Exemplo:

```
100000L
```

• um 'ul' ou 'UL' para forçar a constante em uma constante longo sem sinal.
Exemplo:

```
32767ul
```

int()

Descrição

Converte o valor para o tipo de dados *int*.

Sintaxe

int(x)

Parâmetros

x: um valor de qualquer tipo

Respostas

int

word

Descrição

Uma palavra armazena um número sem sinal de 16 bits, de 0 a 65535. O mesmo que um inteiro sem sinal (*unsigned int*).

Exemplo

```
word w = 10000;
```

long

Descrição

Variáveis longas são variáveis de tamanho estendido para armazenamento de números, e armazenam 32 bits (4 bytes), de -2.147.483.648 a 2.147.483.647.

Se fazendo matemática com números inteiros, pelo menos um dos números tem de ser seguido por um L, obrigando-o a ser longo. Veja na página de constantes inteiras para mais detalhes.

Exemplo

```
long speedOfLight = 186000L;  // veja na página de constantes inteiras para explanação do "L"
```

Sintaxe

```
long var = val;
```

- var – o nome da variável long
- val – o valor declarado na variável

unsigned long

Descrição

Variáveis longas sem sinal (*unsigned long*) são variáveis de tamanho estendido para armazenamento de números, e armazena 32 bits (4 bytes). Ao contrário do longos padrão o longo sem sinal não irá armazenar números negativos, tornando a sua faixa 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();
  //exibe o tempo desde a inciação do programa
  Serial.println(time);
  // espera um segundo para não enviar uma massiva quantidade de dados
  delay(1000);
}
```

Sintaxe

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

- var – o nome a sua variável long
- val – o valor declarado na variável

long()

Descrição

Converte o valor do tipo de dados *long*.

Sintaxe

```
long(x)
```

Parâmetros

x: um valor de qualquer tipo

Respostas

long

short

Descrição

Um curto (*short*) é um tipo de dados de 16 bits. Em todos os Arduinos (baseados em ATmega e ARM) o curto armazena um valor de 16 bits (2 bytes). Isto origina uma gama de 32.768 a 32.767 (valor mínimo de 2 ^ 15, e um valor máximo de (2 ^ 15) - 1).

Exemplo

```
short ledPin = 13;
```

Sintaxe

```
short var = val;
```

- var – o nome da sua variável *short*
- val – o valor declarado na variável

double

Descrição

Número de ponto flutuante de precisão dupla. Para as placas baseadas no Uno e outras Atmega, ocupa 4 bytes. Isto é, a aplicação dupla é exactamente o mesmo que o *floats*, com nenhum ganho em termos de precisão.

Para o Arduino Due, duplas têm precisão de 8 bytes (64 bits).

Dica

Os usuários que pegam código de outras fontes, que incluem variáveis duplas podem querer examinar o código para ver se a precisão implícita é diferente do que realmente alcançado em Arduinos baseados em ATMEGA.

string

Descrição

Seqüências (*strings*) de texto podem ser representadas de duas formas. Você pode usar o tipo de dados *String*, que faz parte do núcleo a partir da versão 0019, ou você pode fazer uma seqüência de uma matriz (*array*) do tipo *char null-terminate* (terminação nula). Esta página descreve o segundo método. Para mais detalhes sobre o objeto *String*, que lhe dá mais funcionalidade ao custo de mais memória, veja a página de objeto *String*.

Exemplos

Todas as seguintes sequencias são declarações válidas.

```
char Str1[15];
char Str2[8] = {'a', 'r', 'd', 'u', 'i', 'n', 'o'};
char Str3[8] = {'a', 'r', 'd', 'u', 'i', 'n', 'o', '\0'};
char Str4[ ] = "arduino";
char Str5[8] = "arduino";
char Str6[15] = "arduino";
```

Possibilidades para declarar sequencias (*strings*)

• Declarar uma matriz de caracteres sem inicializa-la como na Str1

• Declarar uma matriz de caracteres (com um caractere adicional) e o compilador irá adicionar o caractere nulo necessário, como em Str2

• adicionar explicitamente o caráter nulo, str3

• Inicializar com uma sequencia constante entre aspas, o compilador irá definir o tamanho da matriz para se ajustar a seqüência constante e um caractere nulo de terminação, Str4

• Inicializar a matriz com um tamanho e sequencia constante explícita, Str5

• Inicializar a matriz, deixando espaço extra para uma seqüência maior, Str6

Terminação nula

Geralmente, as sequencias são terminadas com um caractere nulo (código ASCII 0). Isso permite que as funções (como *Serial.print()*) digam onde é a extremidade da sequencia. Caso contrário, eles iriam continuar lendo bytes subseqüentes de memória que não são realmente parte da sequencia.

Isto significa que a seqüência precisa ter espaço para mais um caractere do que o texto que você quer contém. É por isso que Str2 e Str5 precisa ter oito caracteres, embora no "arduino" é de apenas sete - a última posição é automaticamente preenchida com um caractere nulo. Str4 será automaticamente dimensionada para oito caracteres, um para o nulo extra. Em str3, incluímos nós mesmos explicitamente o caractere nulo (escrito '\0').

Note que é possível ter uma sequencia sem um caractere nulo final (por exemplo, se você tivesse especificado o comprimento do Str2 como sete em vez de oito). Isso vai quebrar a maioria das funções que usam *strings*, então você não deve fazê-lo intencionalmente. Se você notar algum comportamento estranho (operando em caracteres não na sequencia), no entanto, este poderá ser o problema.

Aspas simples ou duplas?

Sequecias (*strings*) são sempre definidas dentro de aspas ("Abc") e os caracteres são sempre definidos dentro de aspas simples ('A').

Envolvendo cadeias longas

Você pode quebrar cadeias longas como esta:

```
char myString[] = "This is the first line"
" this is the second line"
" etcetera";
```

Matrizes de sequencias

Muitas vezes, é conveniente, quando se trabalha com grandes quantidades de texto, como um projeto com um display LCD, configurar uma matriz de sequencias. Porque as próprias sequencias são matrizes, isto é verdade, em um exemplo de uma matriz bidimensional.

No código abaixo, o asterisco após o tipo de dados *char* *"char*"* indica que se trata de um conjunto de "pontos" (*"pointers"*). Todos os nomes de matrizes são, na verdade, pontos, de modo que este é necessário para fazer uma matriz de matrizes (*array* of *arrays*). Os pontos são uma das partes mais esotérica de C para que os iniciantes entendam, mas não é necessário entender pontos em detalhe para usá-los de forma eficaz aqui.

Exemplo

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

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

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

String (objeto)

Descrição

A classe *String*, parte do núcleo a partir da versão 0019, permite que você use e manipule seqüências de texto em formas mais complexas do que as matrizes de caracteres (*character arrays*). Você pode concatenar *Strings*, acrescentar a elas, procurar e substituir subsequências (substrings) e muito mais. É preciso mais memória do que uma matriz de caracteres simples, mas também é mais útil.

Para referência, matrizes de caracteres são chamados de *strings* com um pequeno s, e as instâncias da classe *String* são referenciados como sequencias com um S capitulado. Note que sequencias constantes, indicadas com

"aspas duplas" são tratadas como matrizes de caracteres, e não instâncias da classe *String*.

Matrizes (Arrays)

Uma matriz é um conjunto de variáveis que são acessadas com um número de índice. Matrizes na linguagem de programação C, em que se baseia o Arduino, pode ser complicada, mas utilizando matrizes simples é relativamente fácil.

Criando (Declarando) uma Matriz

Todos os métodos a seguir são formas válidas de criar (declarar) uma matriz.

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

Você pode declarar uma matriz sem inicializa-la como na *myInts*.

Em *myPins* nós declaramos uma matriz sem escolher explicitamente um tamanho. O compilador conta os elementos e cria uma matriz de tamanho apropriado.

Finalmente, você pode tanto inicializar e definir o tamanho da sua matriz (*array*), como em mySensVals. Note-se que quando se declara uma matriz do tipo *char*, mais um elemento do que a sua inicialização é necessária, para manter o caráter nulo necessário.

Acessando uma Matriz

As matrizes são indexadadas em zero, isto é, referindo-se a inicialização de matriz acima, o primeiro elemento da matriz está no índice 0, daí:

```
mySensVals[0] == 2, mySensVals[1] == 4, and so forth.
Isso também significa que em uma matriz com dez elementos, o índice de nove é o último elemento. Por isso:
int myArray[10]={9,3,2,4,3,2,7,8,9,11};
    // myArray[9]    contém 11
    // myArray[10]   é válida e contém informação aleatória (outro endereço de memória)
```

Por esta razão, você deve ter cuidado ao acessar matrizes (*arrays*). Acessando após o final de uma matriz (usando um número de índice maior do que o tamanho da matriz declarada - 1) é a leitura da memória que está em uso para outros fins. Leitura a partir desses locais, provavelmente, não vai fazer muito, exceto render dados inválidos. Escrevendo nos locais de memória aleatórios é definitivamente uma má idéia e muitas vezes pode levar a resultados infelizes, como falhas ou mau funcionamento do programa. Isso também pode ser um erro difícil de rastrear.

Ao contrário do BASIC ou JAVA, o compilador C não verifica para ver se o acesso à matriz está dentro dos limites legais do tamanho da matriz que você declarou.

Para declarar um valor para matriz:

```
mySensVals[0] = 10;
```

Para recuperar um valor e uma matriz:

```
x = mySensVals[4];
```

Matrizes e ciclos FOR (FOR loops)

As matrizes são frequentemente manipuladas dentro de ciclos (loops), em que o contador de ciclo é utilizado como o índice para cada elemento da matriz. Por exemplo, para exibir os elementos de uma matriz através da porta serial, você poderia fazer algo como isto:

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

Exemplo

Para um programa completo que demonstra o uso de matrizes, consulte o Exemplo Knight Rider dos Tutoriais.

Conversão

char()

Descrição

Converte o valor para o tipo de dados *char*.

Sintaxe

```
char(x)
```

Parâmetros

x: o valor de qualquer tipo

Respostas

char

byte

Description

Um byte armazena um número sem sinal (unsigned number) de 8-bit, de 0 a 255.

Exemplo

```
   byte b = B10010;  // "B" ´o formatador binário formatter (B10010 = 18 decimal)
```

byte()

Descrição

Converte o valor para o tipo de dados byte.

Sintaxe

```
byte(x)
```

Parâmetros

x: um valor de qualquer tipo

Respostas

byte

word

Descrição

Uma palavra armazena um número sem sinal (*unsigned word*) de 16 bits, de 0 a 65535. O mesmo que um inteiro sem sinal (*unsigned int*).

Exemplo

```
word w = 10000;
```

word()

Descrição

Converte o valor do tipo de dados *words* ou cria palavras de dois bytes.

Sintaxe

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

Parâmetros

x: um valor de qualquer tipo
h: o byte maior da ordem (mais a esquerda) da palavra
l: o byte menor da ordem (mais a direita) da palavra

Respostas

word

float

Descrição

Tipo de dados para números de pontos flutuantes (floating-point), um número que tem um ponto decimal. Números de ponto flutuante são muitas vezes utilizados para aproximar os valores analógicos e contínuos, porque eles têm maior resolução do que números inteiros. Números de ponto flutuante pode ser tão grande quanto 3,4028235E+38 e tão baixo quanto -3,4028235+38. Eles são armazenados com 32 bits (4 bytes) de informações.

Floats tem apenas 6-7 dígitos decimais de precisão. Isso significa que o número total de dígitos, e não o número à direita do ponto decimal. Ao contrário de outras plataformas, onde você pode obter mais precisão por meio de um duplo (por exemplo, até 15 dígitos), no Arduino, duplo é o mesmo tamanho de um *float*. Números de ponto flutuante não são exatos, e podem produzir resultados estranhos quando comparados. Por Exemplo 6.0/3.0 pode não ser igual 2.0. É, ao invés deve-se verificar que o valor absoluto da diferença entre o número é menor do que um número pequeno.

Matemática de ponto flutuante também é muito mais lenta do que a matemática de inteiros na realização de cálculos, por isso deve ser evitada se por exemplo, um ciclo tem de ser executado em alta velocidade para uma função com tempo crítico. Os programadores muitas vezes vão a alguns comprimentos para converter cálculos de ponto flutuante para matemática de inteiros para aumentar a velocidade.

Se fazer matemática com *floats*, você precisa adicionar um ponto decimal, caso contrário, ele será tratado como um int. Veja a página de constantes de ponto flutuante para mais detalhes.

Exemplos

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

Sintaxe

```
float var = val;
```

- var – o nome da sua variável *float*
- val – o valor declarado na variável

Exemplo de Código

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

x = 1;
y = x / 2;          // y agora contém 0, ints não pode conter frações
z = (float)x / 2.0;   // z agora contém 5 (você precisa usar 2.0, não 2)
```

float()

Descrição

Converte o valor o tipo de dados *float*.

Sintaxe

```
float(x)
```

Parâmetros

x: um valor de qualquer tipo

Respostas

float

Notas

Consulte a referência de *float* para obter detalhes sobre a precisão e as limitações de números de ponto flutuante no Arduino.

Variável Escopo & Qualificações

Variável Escopo

Variáveis na linguagem de programação C, que o Arduino usa, tem uma propriedade chamada escopo (*scope*). Isso é um contraste com as primeiras versões de linguagens como BASIC onde cada variável é uma variável global.

Uma variável global é aquela que pode ser vista por cada função de um programa. As variáveis locais são visíveis apenas para a função em

que são declaradas. No ambiente Arduino, qualquer variável declarada fora de uma função (por exemplo, setup (), loop (), etc), é uma variável global.
Quando os programas começam a ficar maiores e mais complexos, variáveis locais são uma maneira útil para garantir que apenas uma função tem acesso a suas próprias variáveis. Isso evita erros de programação quando uma função inadvertidamente modifica variáveis utilizadas por outra função.
Também é por vezes útil para declarar e inicializar uma variável dentro de um ciclo *for*. Isso cria uma variável que só pode ser acessado a partir de dentro do ciclo-for entre colchetes.

Exemplo:

```
int gPWMval;  // qualquer função irá ver a esta varíavel

void setup()
{
  // ...
}

void loop()
{
  int i;    // "i" é somente “visível” dentro do “ciclo”
  float f;  // "f" é somente “visível” dentro do “ciclo”
  // ...

  for (int j = 0; j <100; j++){
  // a variável j pode apeas ser acessada de dentro dos colchetes do ciclo-for
  }

}
```

Static

A palavra-chave estático (*static*) é usada para criar variáveis que são visíveis a uma única função. No entanto, ao contrário de variáveis locais que são criadas e destruídas cada vez que uma função é chamada, variáveis estáticas persistem além da chamada de função, preservando seus dados entre chamadas de função.
Variáveis declaradas como estatícas só serão criadas e inicializadas pela primeira vez que uma função for chamada.

Exemplo

```
/ * RandomWalk
* Paul Badger 2007
* RandomWalk vagueia de cima para baixo ao acaso entre dois pontos. O movimento máximo de um ciclo é regido pelo parâmetro “tamanho do passo” ("stepsize").
* Uma variável estática é movida para cima e para baixo numa quantidade aleatória.
* Esta técnica também é conhecida como "ruído rosa" e "pé bêbado".
* /

#define randomWalkLowRange -20
#define randomWalkHighRange 20
int stepsize;

int thisTime;
int total;

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

void loop()
{       //  testa função randomWalk
  stepsize = 5;
  thisTime = randomWalk(stepsize);
  Serial.println(thisTime);
   delay(10);
}

int randomWalk(int moveSize){
  static int  place;     // variável para armazenar o valor no caminho aleatório - declarada estática para que ele armazene
// Valores entre chamadas de função, mas outras funções não podem alterar o seu valor

  place = place + (random(-moveSize, moveSize + 1));

  if (place < randomWalkLowRange){                  // check lower and upper limits
    place = place + (randomWalkLowRange - place);     // reflect number back in positive direction
  }
  else if(place > randomWalkHighRange){
    place = place - (place - randomWalkHighRange);     // reflect number back in negative direction
  }

  return place;
}
```

Palavra-chave volatile

Volátil (*volatile*) é uma palavra-chave conhecida como um qualificador variável, é geralmente usado antes do tipo de dados de uma variável, para modificar a maneira pela qual o compilador e programa subsequente trata a variável.
Declarar uma variável volátil é uma diretiva do compilador. O compilador é um software que traduz o seu código C/C++ para o código da máquina, que são as instruções reais para o chip Atmega do Arduino.
Especificamente, ele orienta o compilador para carregar a variável a partir da RAM e não a partir de um registador de armazenamento, o que é uma localização de memória temporária em que as variáveis do programa são

armazenadas e manipuladas. Sob certas condições, o valor de uma variável armazenada em registos pode ser imprecisa.
Uma variável pode ser declarada volátil sempre que o seu valor puder ser alterado por algo além do controle da seção de código em que aparece, como uma linha em execução simultânea. No Arduino, o único lugar provável que isso ocorra é em seções de código associados com interrupções, chamado de rotina de interrupção do serviço.

Exemplo

```
// alterna o LED quando o pino de interrupção muda de estado

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;
}
```

Palavra-chave const

A palavra-chave *const* significa constante. É um qualificador variável que modifica o comportamento da variável, tornando uma variável em "somente leitura". Isto significa que a variável pode ser utilizada, assim como qualquer outra variável desse tipo, mas o seu valor não pode ser alterado. Você receberá um erro do compilador se você tentar atribuir um valor a uma variável constante.
Constantes definidas com a palavra-chave *const* obedecem as regras de escopo de variáveis que governam outras variáveis. Isso, e as armadilhas do uso do *#define*, faz a palavra-chave *const* um método superior para a definição de constantes e é preferível a usar *#define*.

Exemplo

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

// ....

x = pi * 2;    // é correto usar const em matemática

pi = 7;        // ilegal – você não pode escrever (modificar) uma constante
```

#define ou const

Você pode usar *const* ou *#define* para criar constantes numéricas ou sequencias. Para matrizes (*arrays*), você vai precisar usar *const*. Em geral *const* é preferível ao *#define* para definir constantes.

Utilidades

sizeof

Descrição

O operador *sizeof* devolve o número de bytes de um tipo variável, ou o número de bytes ocupados por uma matriz.

Sintaxe

sizeof(variable)

Parâmetros

variable: qualquer tipo de varíavel ou matriz (exemplo *int*, *float*, *byte*)

Exemplo de código

O operador *sizeof* é útil para o tratamento das matrizes (tais como sequencias), quando é conveniente para ser capaz de mudar o tamanho da matriz sem quebrar outras partes do programa.
Este programa exibe uma seqüência de texto, um carácter de cada vez. Tente mudar a frase do texto.

```
char myStr[] = "this is a test";
int i;

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

void loop() {
  for (i = 0; i < sizeof(myStr) - 1; i++){
    Serial.print(i, DEC);
    Serial.print(" = ");
    Serial.write(myStr[i]);
    Serial.println();
  }
  delay(5000);  // desacela o programa
}
```

Note que *sizeof* retorna o número total de bytes. Então para tipos de varíaveis longas como Inteiros (*ints*), o ciclo seria algo parecido com isso. Note também que uma sequencia

formatada corretamente termina com o símbolo *NULL*, que tem valor ASCII 0.

```
for (i = 0; i < (sizeof(myInts)/sizeof(int)) - 1; i++) {
  // faz alguma coisa com myInts[i]
}
```

Funções

Digital I/O

Digital Pins (Pinos Digitais)

Os pinos no Arduino podem ser configurados como entradas ou saídas. Este documento explica o funcionamento dos pinos nos dois modos. Enquanto o título deste documento refere-se aos pinos digitais, é importante notar que a grande maioria dos pinos analógicos no Arduino (Atmega), podem ser configurados e utilizados, exatamente da mesma maneira que os pinos digitais.

Propriedades de pinos configurados como entrada

O padrão dos pino no Arduino (Atmega) é para entradas, então eles não precisam ser explicitamente declarados como entradas (*INPUT*) no *pinMode()* quando você estiver usando-os como entradas. Pinos configurados desta forma são ditos em um estado de alta impedância. Pinos de entrada fazem extremamente pequenas exigências sobre o circuito de amostragem, equivalente a uma resistência em série de 100 megaohm em frente do pino. Isso significa que é preciso muito pouca corrente para mover o pino de entrada de um estado para outro, e pode tornar os pinos úteis para tarefas como a implementação de um sensor de toque capacitivo, lendo um LED como um fotodiodo , ou ler um sensor analógico com um esquema de RCTime .

Isto também significa no entanto, que os pinos configurados como *pinMode(pin, INPUT)* com nada ligado a eles, ou com os fios conectados a eles que não estão ligados a outros circuitos , irão relatar mudanças aparentemente aleatórias no estado do pino, a captação de ruído elétrico do ambiente, ou o acoplamento capacitivo do estado de um pino perto.

Resistores Pullup com pinos configurados como entrada (INPUT)

Muitas vezes é útil orientar um pino de entrada para um estado conhecido, se nenhuma entrada estiver presente. Isto pode ser feito através da adição de uma resistência pull-up (a +5V), ou de uma resistência pull-down (resistência para o terra) na entrada. Um resistor de 10K é um bom valor para um resistor pull-up ou pull-down.

Propriedades dp Pino configurado como INPUT_PULLUP

Há resistores pull-up de 20k embutido no chip Atmega que podem ser acessados a partir de software. Estes resistores pull-up internos são acessados, definindo o *pinMode()* como *INPUT_PULLUP* . Isso inverte efetivamente o comportamento do modo de entrada, onde *HIGH* significa que o sensor está desligado, e *LOW* significa que o sensor está ligado.

O valor deste pull-up depende do microcontrolador utilizado. Na maioria das placas baseadas em AVR, o valor é garantido para estar entre 20kΩ e 50kΩ. Para o Arduino Due, é entre 50kΩ e 150kΩ. Para o valor exato, consulte a folha de dados do microcontrolador da sua placa.

Ao ligar um sensor a um pino configurado como *INPUT_PULLUP*, a outra extremidade deve ser ligada ao terra. No caso de um interruptor simples, isso faz com que o pino leia *HIGH* quando o interruptor estiver aberto, e *LOW* quando o interruptor é pressionado.

Os resistores pull-up fornecem corrente suficiente para acender um LED fracamente conectado a um pino que foi configurado como uma entrada. Se os LEDs em um projeto parecerem estar funcionando, mas muito fracos, é provável que é isto que está acontecendo.

Os resistores pull-up são controlados pelos mesmos registradores (locais de memória interna do chip) que controlam se um pino é *HIGH* ou *LOW*. Consequentemente, um pino que está configurado para ter ligado o resistor pull-up quando o pino for uma entrada, terá o pino configurado como *HIGH* se o pino estiver ligado a uma saída com *pinMode()*. Isso funciona na outra direção também, e um pino de saída que é deixado em um estado *HIGH* terá as resistências de pull-up definidas para entrada com *pinMode()*.

Antes do Arduino 1.0.1, era possível configurar os pull-ups internos da seguinte maneira:

```
pinMode(pin, INPUT);           // define o pino como entrada
digitalWrite(pin, HIGH);       // liga os resistores pull-up
```

NOTA: O pino digital 13 é mais difícil de usar como uma entrada digital do que os outros pinos digitais, porque tem um LED e um resistor

ligado a ele, que é soldado à placa na maioria das placas. Se você habilitar o resistor pull-up interno de 20k, ele vai travar em torno de 1.7V em vez do 5V esperado porque o LED e resistor em série puxará o nível de tensão para baixo, o que significa que sempre retorna *LOW*. Se você dever usar o pino 13 como uma entrada digital, defina seu *pinMode()* para entrada e use um resistor pull down externo.

Propriedades dos Pinos Configurados como OUTPU (Saída)

Pinos configurados como *OUTPUT* com *pinMode()* são ditos em um estado de baixa impedância. Isto significa que eles podem fornecer uma quantidade substancial de corrente para outros circuitos. Pinos Atmega podem prover (fornecer corrente positiva) ou ceder (fornecer corrente negativa) até 40 mA (miliamperes) de corrente para outros dispositivos/circuitos. Esta corrente é suficiente para a luz do LED brilhar (não se esqueça do resistor em série), ou executar muitos sensores, por exemplo, mas não o suficiente para executar a maioria dos relés, solenóides, ou motores.

Circuitos curtos nos pinos do Arduino, ou tentar executar dispositivos de alta corrente com eles, pode danificar ou destruir os transistores de saída no pino, ou danificar todo o chip Atmega. Muitas vezes, isso irá resultar em um pino "morto" no microcontrolador, mas o chip restante continuará a funcionar de forma adequada. Por esta razão, é uma boa idéia conectar os pinos de SAÍDA para outros dispositivos com resistores de 470Ω ou de 1k, ao menor corrente máxima dos pinos é necessário para uma determinada aplicação.

pinMode()

Descrição

Configura o pino especificado a se comportar tanto como uma entrada (*input*) ou uma saída(*output*). Consulte a descrição de pinos digitais para obter detalhes sobre a funcionalidade dos pinos.

A partir do Arduino 1.0.1, é possível ativar os resistores pull-up internos com o modo *INPUT_PULLUP*. Além disso, o modo de entrada desativa explicitamente os pullups internos.

Sintaxe

pinMode(pin, mode)

Parâmetros

pin: o número do pino cujo modo você deseja definir

mode: INPUT, OUTPUT, or INPUT_PULLUP. (veja a página de pinos digitais (digital pins) para uma descrição mais completa das funcionalidades.

Respostas

Nenhum

Exemplo

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

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

void                      loop()
{
  digitalWrite(ledPin, HIGH);   // define o LED como ligado
   delay(1000);                 // espera um segundo
   digitalWrite(ledPin, LOW);   // define o LED como desligado
   delay(1000);                 // espera um segundo
}
```

Nota

Os pinos análogicos de entrada podem ser usados como pinos digitais, referenciados como A0, A1, etc.

digitalWrite()

Descrição

Escreva um valor *HIGH* ou *LOW* para um pino digital.

Se o pino foi configurado como uma saída (*OUTPUT)* com *pinMode()*, a tensão será definida com o valor correspondente : 5V (ou 3.3V em placas de 3.3V) em *HIGH* , 0V (terra) para *LOW*.

Se o pino for configurado como entrada, escrevendo o valor *HIGH* com *digitalWrite()* irá permitir um resistor pull-up interno de 20K (veja o tutorial sobre pinos digitais). Escrevendo *LOW* irá desativar o pull-up. O resistor pull-up é suficiente para acender um LED fraco, por isso, se LEDs aparecem para trabalhar, mas muito fracos, esta é a causa provável. O remédio é definir o pino como uma saída com a função pinMode ().

NOTA: o pino digital 13 é mais difícil de usar como uma entrada digital do que os outros pinos digitais, porque ele tem um LED e um resistor ligado a ele, que são soldados à placa,

na maioria das placas. Se você habilitar o seu resistor pull-up interno de 20K, ele vai travar em torno de 1,7 V em vez do 5V esperado porque o LED onboard e resistor em série puxa o nível de tensão para baixo, o que significa que sempre retorna *LOW*. Se você deve usar o pino 13 como uma entrada digital, utilize um resistor pull-down externo.

Sintaxe

digitalWrite(pin, value)

Parâmetros

pin: o número do pino
value: *HIGH* ou *LOW*

Respostas

nenhum

Exemplo

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

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

void loop()
{
  digitalWrite(ledPin, HIGH);   // define o LED como ligado
  delay(1000);                  // espera um segundo
  digitalWrite(ledPin, LOW);    // define o LED como desligado
  delay(1000);                  // espera um segundo
}
```

Definindo o pino 13 para *HIGH*, faz um atrado longo de um segundo, e define o pino de volta para *LOW*.

Nota

Os pinos de entrada análogicos podem ser usados como pinos digitais, referenciando como A0, A1, etc.

digitalRead()

Descrição

Lê o valor de um pino digital específico, ambos *HIGH* ou *LOW*.

Sintaxe

digitalRead(pin)

Parâmetros

pin: o número do pinp digital que você quer que leia (*int*)

Respostas

HIGH ou *LOW*

Exemplo

Define o pino 13 com o mesmo valor do pino 7, declarados como entrada.

```
int ledPin = 13; // LED conectado ao pino digital 13
int inPin = 7;   // botão de pressão conectado ao pino digital 7
int val = 0;     // variável armazenada no valor de leitura

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

void loop()
{
   val = digitalRead(inPin);   // lê o pino de entrada
  digitalWrite(ledPin, val);   // define o LED para o valor do botão
}
```

Nota

Se o pino não estiver conectado a nada, *digitalRead()* pode retornar tanto *HIGH* ou *LOW* (e isto pode mudar randomicamente).

Os pinos análogicos de entrada podem ser usados como pinos digitais, referenciando como A0, A1, etc.

Analog I/O

Pino Analógico de Entrada

A descrição dos pinos analógicos de entrada no chip Arduino (Atmega8, Atmega168, Atmega328, ou Atmega1280).

Conversor A/D

Os controladores Atmega utilizados no Arduino contém um conversor onboard de 6 canais analógicos-para-digitais (A/D). O conversor tem uma resolução de 10 bits, retornando números inteiros de 0 a 1023. Enquanto a principal função dos pinos analógicos para a maioria dos usuários do Arduino é ler sensores analógicos, os pinos analógicos também têm todas as funcionalidades de uso geral de pinos de entrada/saída (GPIO) (o mesmo que pinos digitais 0-13).

Consequentemente, se um usuário necessita de mais funcionalidades de uso geral de pinos de entrada-saída, e todos os pinos analógicos não estiverem em uso, os pinos analógicos podem ser utilizados para GPIO.

Mapeamento do Pino

Os pinos analógicos podem ser usados de forma idêntica aos pinos digitais, usando os pseudônimos A0 (para entrada analógica 0), A1, etc. Por exemplo, o código ficaria assim

para definir o pino analógico 0 para uma saída, e configurá-lo como *HIGH*:

```
pinMode(A0, OUTPUT);
digitalWrite(A0, HIGH);
```

Resistores Pull-up

Os pinos analógicos também têm resistores pull-up, que funcionam de forma idêntica aos resistores pull-up dos pinos digitais. Eles são ativados através do envio de um comando como:

```
digitalWrite(A0, HIGH);  // set pull-up on analog pin 0
```

Enquanto o pino é uma entrada.
Esteja ciente, porém, que acender um pull-up afetará os valores relatados pelo *analogRead()*.

Detalhes e Advertências

O comando *analogRead* não irá funcionar corretamente se um pino tiver sido previamente definido como uma saída, por isso, se este for o caso, defina-o de volta como uma entrada antes de usar *analogRead*. Da mesma forma, se o pino foi definido para uma saída com *HIGH*, o resistor pull-up será definido, quando o interruptor voltar a ser uma entrada.
A folha de dados do Atmega também adverte contra a mudança de pinos analógicos em estreita proximidade temporal para fazer leituras A/D (*analogRead*) em outros pinos analógicos. Isto pode causar ruído elétrico e introduzir instabilidade ao sistema analógico. Pode ser desejável, depois de manipular os pinos analógicos (no modo digital), adicionar um pequeno atraso antes de usar *analogRead()* para ler outros pinos analógicos.

analogReference(type)

Descrição

Configura a tensão de referência utilizada para a entrada analógica (ou seja, o valor utilizado como a parte superior do intervalo de entrada). As opções são:

- DEFAULT (padrão): a referência analógica padrão de 5 volts (em placas Arduino de 5V) ou 3,3 volts (em placas Arduino 3.3V)
- INTERNAL (interno): embutida de referência, igual a 1,1 volts no ATmega168 ou ATmega328 e 2,56 volts no ATmega8 (não disponível no Arduino Mega)
- • INTERNAL1V1: uma referência 1.1V embutida (Arduino mega apenas)
- • INTERNAL2V56: uma referência 2.56V embutida (Arduino mega apenas)
- • EXTERNAL (externa): a tensão aplicada ao pino AREF (0 a 5V apenas) é usado como a referência.

Parâmetros

type: cada tipo de referência a ser usada (DEFAULT,INTERNAL, INTERNAL1V1, INTERNAL2V56, ou EXTERNAL).

Respostas

Nenhuma.

Nota

Depois de alterar a referência analógica, as primeiras leituras de *analogRead()* podem não ser precisas.

Cuidados

Não use nada menor de 0V ou maior que 5V para a tensão de referência externa no pino AREF! Se você estiver usando uma referência externa no pino AREF, você deve definir a referência analógica para *EXTERNAL* antes de chamar o *analogRead()*. Caso contrário, haverá curto juntos a tensão de referência ativa (gerada internamente) ao pino AREF, podendo danificar o microcontrolador da sua placa Arduino.
Alternativamente, você pode ligar a tensão de referência externa para o pino AREF através de um resistor de 5K, que lhe permite alternar entre tensões de referência externas e internas. Note que a resistência vai alterar 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, 2,5V aplicada através da resistência irá produzir 2,5 * 32 / (32 + 5) = ~ 2,2 V, no pino AREF.

analogRead()

Descrição

Lê o valor do pino analógico especificado. A placa Arduino contém 6 canais (8 canais no Mini e Nano, 16 na Mega), conversor analógico-digital de 10-bit. Isso significa que ele irá mapear tensões de entrada entre 0 e 5 volts em valores inteiros entre 0 e 1023. Isso gera uma resolução entre as leituras de: 5 volts/1024 unidades, ou 0,0049 volts (4,9 mV) por unidade. O alcance e a resolução de entrada pode ser alterado usando *analogReference()*.
Demora cerca de 100 microssegundos (0,0001 s) 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 do pino de entrada analógica para ler a partir de (0 a 5 na maioria das placas, 0 a 7 no Mini e Nano, 0 a 15 na Mega)
Respostas
int (0 a 1023)
Nota
Se o pino de entrada analógica não está conectado a nada, o valor retornado pelo *analogRead()* irá variar com base em uma série de fatores (por exemplo, os valores das outras entradas analógicas, quão perto sua mão está da placa, etc.).
Exemplo

```
int analogPin = 3;        // Limpador de potenciômetro (terminal central) ligado ao pino analógico 3

                          // Exterior leva à terra e +5 V

int val = 0;              // variável para armazenar o valor lido


void                                  setup()

{

   Serial.begin(9600);                // configura o serial

}



void                                  loop()

{

  val = analogRead(analogPin);    // lê o pino de entrada

   Serial.println(val);               // depura o valor

}
```

analogWrite()
Descrição
Grava um valor analógico (onda PWM) a um pino. Pode ser usado para acender um LED com diferentes brilhos ou acionar um motor em várias velocidades. Após uma chamada para *analogWrite()*, o pino irá gerar uma onda quadrada constante do ciclo de trabalho especificado até a próxima chamada para *analogWrite()* (ou uma chamada para *digitalRead()* ou *digitalWrite()* no mesmo pino). A freqüência do sinal PWM na maioria dos pinos é de cerca de 490 Hz. No Uno e placas semelhantes, os pinos 5 e 6 têm uma frequência de aproximadamente 980 Hz. Os pinos 3 e 11 no Leonardo também operam a 980 Hz.
Na maioria das placas Arduino (aquelas com ATmega168 ou ATmega328), essa função funciona nos pinos 3, 5, 6, 9, 10 e 11. No Arduino Mega, ele funciona nos pinos 2 - 13 e 44-46. Placas Arduino mais antigas com um ATmega8 só suportam *analogWrite()* nos pinos 9, 10 e 11.
O Arduino Due suporta *analogWrite()* nos pinos de 2 a 13, além dos pinos DAC0 e DAC1. Ao contrário dos pinos PWM, DAC0 e DAC1 são conversores digitais para analógicos e agem como saídas analógicas verdadeiras.
Você não precisa chamar *pinMode()* para definir o pino como uma saída antes de chamar *analogWrite()*.
A função *analogWrite* tem nada a ver com os pinos analógicos ou a função *analogRead*.
Sintaxe
analogWrite(pin, value)
Parâmetros
pin: o pino que vai escreve o.
value: O ciclo de trabalho: entre 0 (sempre desligado) e 255 (sempre ligado).
Respostas
Nenhuma
Notas e problemas conhecidos
As saídas PWM geradas nos pinos 5 e 6 terão ciclos de trabalho mais elevados do que o esperado. Isto é devido à interação com as funções *millis()* e *delay()*, que compartilham o mesmo temporizador interno usado para gerar essas saídas PWM. Este será notado principalmente em baixas configurações de ciclo de trabalho (por exemplo, 0-10) e pode resultar que um valor de 0 não fique totalmente desligado na saída nos pinos 5 e 6.
Exemplo
Define a saída para o LED proporcional ao valor lido a partir do potenciômetro.

```
int ledPin = 9;       // LED conectado ao pino digital 9

int analogPin = 3;    // potenciometro conecatado ao pino analógico 3

int val = 0;          // variável armazenada no valor de leitura


void                                  setup()

{
```

```
  pinMode(ledPin, OUTPUT);  // define o pino como saída
}

void loop()
{
  val = analogRead(analogPin);   // Lê o pino de entrada
  analogWrite(ledPin, val / 4);  // o valor do analogRead de 0 a 1023, o valor de analogWrite de 0 a 255
}
```

I/O Avançado

tone()

Descrição

Gera uma onda quadrada de freqüência especificada (em 50% do ciclo de trabalho) em um pino. A duração pode ser especificada, caso contrário, a onda continua até que uma chamada para *noTone()*. O pino pode ser ligado a um sirene piezoelétrica ou a outro auto-falante para reproduzir tons.

Apenas um tom pode ser gerado de cada vez. Se um tom já está tocando em um pino diferente, a chamada para *tone()* não terá nenhum efeito. Se o tom está tocando no mesmo pino, a chamada irá definir a sua frequência.

Uso da função *tone()* irá interferir com as saída PWM nos pinos 3 e 11 (em outras placas que não o mega).

Não é possível gerar tons mais baixos do que 31Hz. Para obter detalhes técnicos, consulte as notas de Brett Hagman's notes.

NOTA: se você quiser tocar diferentes notas em vários pinos, você precisa chamar *noTone()* em um pino antes de chamar *tone()* no próximo pino.

Sintaxe

tone(pino, frequencia)
tone(pino, frequencia, duração)

Parâmetros

pin: o pino com o qual irá gerar o tom

frequencia: a frequência do tom em hertz – inteiro sem sinal (*unsigned int*)

duração: a duração do tom e milissegundos (opcional) – longo sem sinal (*unsigned long*)

Respostas

Nenhuma

noTone()

Descrição

Pára a geração de uma onda quadrada desencadeada por *tone()*. Não tem nenhum efeito se não houver tom sendo gerado.

NOTA: se você quiser tocar diferentes notas em vários pinos, você precisa chamar *noTone()* em um pino antes de chamar *tone()* no próximo pino.

Sintaxe

noTone(pino)

Parâmetros

pin: o pino no qual você quer deixar de gerar o tom

Respostas

nenhum

shiftOut()

Descrição

Muda um byte de dados, um bit de cada vez. Começa a partir de ambos os bits mais significantes (ou seja, o mais à esquerda) ou menos (mais à direita). Cada bit será escrito por vez, em pino de dados, após o qual um pino de relógio é pulsado (tornando alto, então baixo) para indicar que o bit está disponível.

Nota: se você está interagindo com um dispositivo que é cronometrado pelo aumento de margens, você precisa ter certeza de que o pino do relógio está em *LOW* antes da chamada para *ShiftOut()*, por exemplo, com uma chamada para *digitalWrite*(*clockPin, LOW*).

Esta é uma implementação de software, ver também a biblioteca SPI, que fornece uma implementação de hardware que é mais rápida, mas só funciona em pinos específicos.

Sintaxe

shiftOut(dataPin, clockPin, bitOrder, valor)

Parâmetros

dataPin: o pino com o qual sairá cada bit (*int*)

clockPin: o pino para alternar uma vez que o dataPin tenha sido definido para o valor correto (int)

bitOrder: mudar os bits; ou MSBFIRST ou LSBFIRST. (Bit mais significativo primeiro, ou bit menos significativo primeiro)

value: os dados para transferir para fora (byte).

Respostas

Nenhuma

Nota

O *dataPin* e *clockPin* já devem estar configurados como saídas por uma chamada para *pinMode()*.

ShiftOut é atualmente escrito para a saída de 1 byte (8 bits), por isso requer uma operação de

duas etapas para valores de saída maiores que 255.

```
// Faz isso para o serial MSBFIRST
int data = 500;
// shift out highbyte
shiftOut(dataPin, clock, MSBFIRST, (data >> 8));
// shift out lowbyte
shiftOut(dataPin, clock, MSBFIRST, data);

// Ou faz isso para o serial LSBFIRST
data = 500;
// shift out lowbyte
shiftOut(dataPin, clock, LSBFIRST, data);
// shift out highbyte
shiftOut(dataPin, clock, LSBFIRST, (data >> 8));
```

Exemplo

Para o circuito de acompanhamento, veja o tutorial sobre como controlar um registo de deslocamento 74HC595.

```
//**************************************************************/
//
// Name : shiftOutCode, Hello World //
// Author : Carlyn Maw,Tom Igoe //
// Date : 25 Oct, 2006 //
// Version : 1.0 //
// Notes : Code for using a 74HC595 Shift Register //
// : to count from 0 to 255 //
//**************************************************************
*

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

void setup() {
  //define os pinos para saída porque eles são abordados no circuito principal
  pinMode(latchPin, OUTPUT);
  pinMode(clockPin, OUTPUT);
  pinMode(dataPin, OUTPUT);
}

void loop() {
  //contador de rotina
  for (int j = 0; j < 256; j++) {
    //terra latchPin e mantém baixo por tanto tempo quanto você está transmitindo digitalWrite(latchPin, LOW);
    shiftOut(dataPin, clockPin, LSBFIRST, j);
    // retorna o latchpin para alto para sinalizar ao chip que ele
    // não precisa mais ouvir informações digitalWrite(latchPin, HIGH);
    delay(1000);
  }
}
```

shiftIn()

Descrição

Muda um byte de dados, um bit de cada vez. Começa a partir de ambos os bits mais significantes (ou seja, o mais à esquerda) ou menos (mais à direita). Para cada bit, o pino do relógio é puxado para cima, o bit seguinte é lido a partir da linha de dados, e, em seguida, o pino do relógio fica de baixo.

Se você está interagindo com um dispositivo que é cronometrado pelo aumento de margens, você precisa ter certeza de que o pino do relógio está baixo antes da primeira chamada para *ShiftIn()*, por exemplo, com uma chamada para *digitalWrite(clockPin, LOW)*.

Nota: Esta é uma implementação de software, ver também a biblioteca SPI, que fornece uma implementação de hardware que é mais rápida, mas só funciona em pinos específicos.

Sintaxe

```
byte incoming = shiftIn(dataPin, clockPin, bitOrder)
```

Parâmetros

dataPin: o pino com o qual entra cada bit (*int*)

clockPin: o pino para mudar o sinal de leitura do dataPin

bitOrder: com o qual muda os bits; ou MSBFIRST ou LSBFIRST (Bit mais significativo primeiro, ou bit menos significativo primeiro)

Respostas

O valor de leitura (*byte*)

pulseIn()

Descrição

Lê um pulso (*HIGH* ou *LOW*) em um pino. Por Exemplo, se o valor é *HIGH*, *pulseIn()* aguarda o pino ser *HIGH*, inicia o tempo, em seguida, aguarda o pino ir para *LOW* e para o tempo. Retorna a duração do pulso, em microssegundos. Aciona e retorna 0 se nenhum pulso começou dentro de um tempo especificado.

O momento desta função foi determinado empiricamente e, provavelmente, vai mostrar os erros em pulsos mais longos. Funciona em pulsos de 10 microssegundos a 3 minutos de duração.

Sintaxe

```
pulseIn(pin, value)
pulseIn(pin, value, timeout)
```

Parâmetros

pin: o número do pino com o qual você quer ler o pulso (*int*).

value: o tipo de pulso a ser lido: ambos HIGH ou LOW. (*int*)

timeout (opcional): o número de microssegundos até o pulso iniciar; o padrão é de um segundo (*unsigned long*)

Respostas

O comprimento do pulso (em microssegundos) ou 0 se se nenhum pulso for iniciado antes do tempo terminar (*unsigned long*)

Exemplo

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

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

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

Segue o passo usando a função *tone()*

Este exemplo mostra como usar o comando *tone()* para gerar uma nota que segue os valores de uma entrada analógica

Hardware Requirido

- Alto-falante de 8 ohms
- 1 fotocélula
- Resistor de 4.7K ohm
- Resitor de 100 ohm
- Placa de ensaio (breadboard)
- Fio de ligação

Circuito

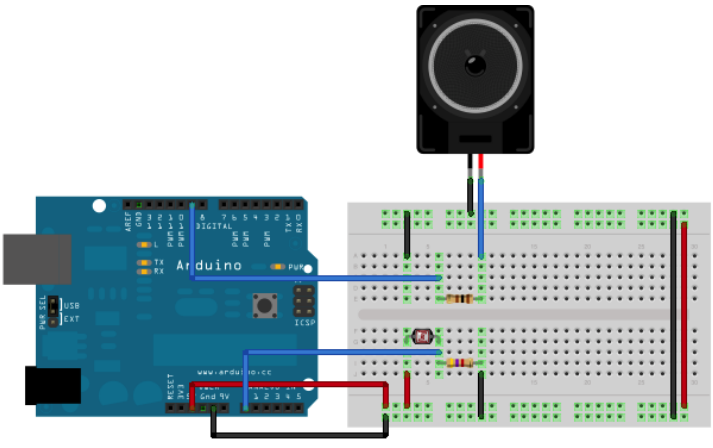
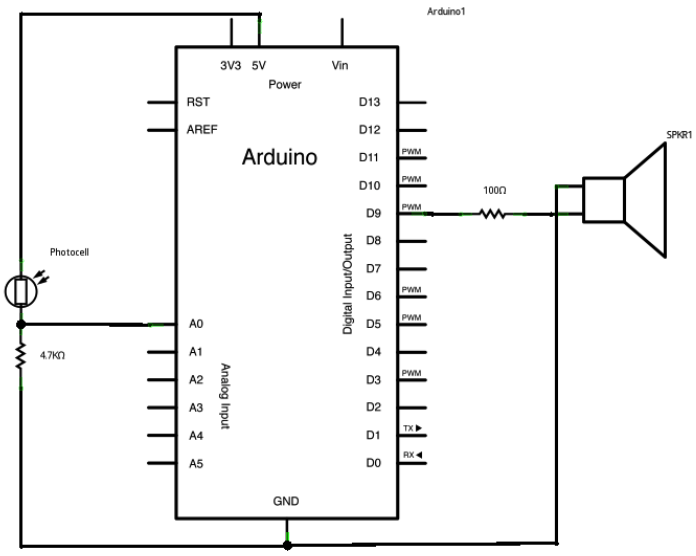


Imagem desenvolvida usando Fritzing. Para mais exemplos de circuitos, consulte a página do projeto Fritzing

Ligue um terminal do seu alto-falante ao pino digital 9 através de um resistor de 100 ohm, e o outro terminal ao terra. Ligue sua fotocélula a 5V, e conecte-a ao pino analógico 0 com a adição de um resistor de 4.7K ao terra.

Esquema



Código

O código para este exemplo é muito simples. Basta pegar uma entrada analógica e mapear seus valores a uma série de tons audíveis. Os seres humanos podem ouvir de 20-20.000Hz, mas 120-1500 geralmente funcionam muito bem para este esquema.

Você vai precisar de ter o alcance real da entrada analógica para o mapeamento. No circuito mostrado, o valor de entrada analógico variou de cerca de 400 a cerca de 1000. Altere os valores do comando *map()* para coincidir com o alcance do sensor.

O esboço é como se segue:

```
/*
 Seguidor de passo

 Reproduz uma nota que muda com base em na
mudança da entrada analógica

 circuito:
 * Alto-falante de 8 ohms no pino digital 9
 * Fotocélula no pino analógico 0 a 5V
 * Resistor 4.7K no pino analógico 0 ao terra

 criado 21 de janeiro de 2010
 modificado 31 mai 2012
 por Tom Igoe, com a sugestão de Michael Flynn

Este exemplo de código é de domínio público.

 http://arduino.cc/en/Tutorial/Tone2

 */


void setup() {
  // inicializa a comunicação serial (apenas para
```

```
depuração):
  Serial.begin(9600);
}

void loop() {
  // lê o sensor:
  int sensorReading = analogRead(A0);
  //exibe a leitura do sensor para ver o alcance
  Serial.println(sensorReading);
  // mapea o alcance da entrada analógica (neste caso, 400 - 1000 para a fotocélula)
  // para o alcance da nota de saída (120 - 1500Hz)
  // altera os números de entrada mínima e máxima abaixo
  // dependendo da faixa de alcance do seu sensor:
  int thisPitch = map(sensorReading, 400, 1000, 120, 1500);

  // toca a nota:
  tone(9, thisPitch, 10);
  delay(1);        // atrasa entre as leituras para estabilidade
}
```

Teclado simples usando a função tone()

Este exemplo mostra como usar o comando *tone()* para gerar diferentes notas dependendo de qual sensor é pressionado.

Hardware Requerido

- Alto-falante de 8 ohms
- (3) Resistores de sensores de força
- (3) resistores de 10k ohm
- (1) 100 ohm resistor
- placa de ensaio (breadboard)
- Fio de conexão

Circuito

Ligue um terminal do seu alto-falante ao pino digital 8 através de um resistor de 100 ohm, e seu outro terminal para o terra.

Alimente seus três FSRs (ou qualquer outro sensor analógico) com 5V em paralelo. Ligue cada sensor com os pinos analógicos 0-2, utilizando uma resistência de 10K, com referência para o terra em cada linha de entrada.

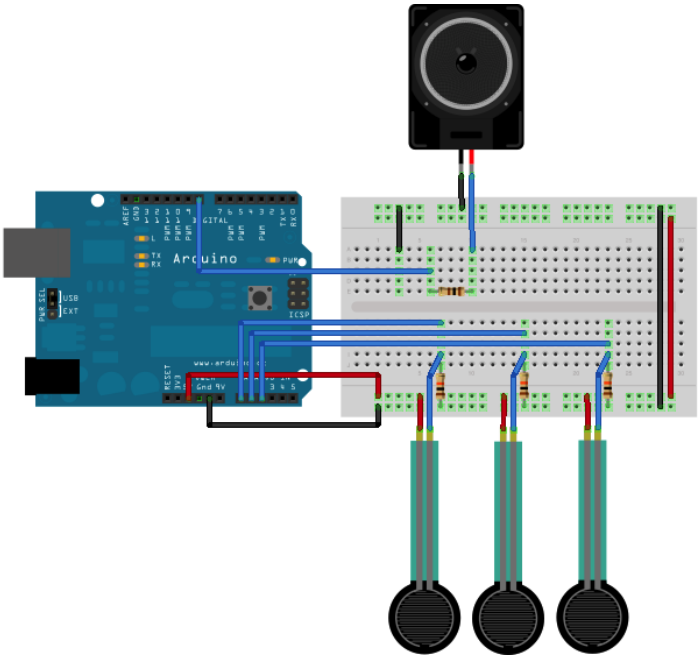
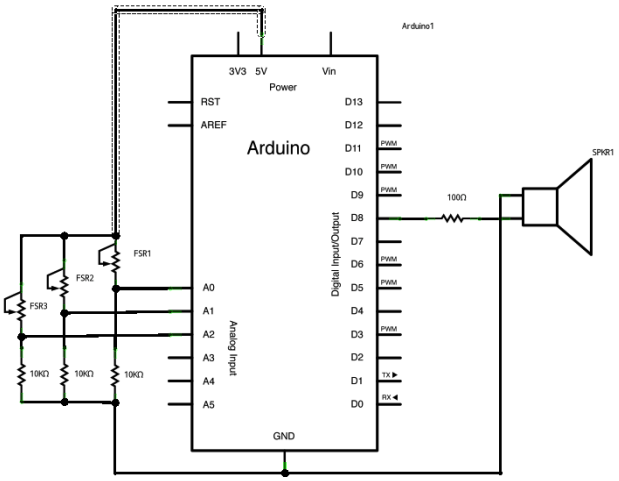


Imagem desenvolvida usando Fritzing. Para mais exemplos de circuitos, consulte a página do projeto Fritzing

Esquema



Código

O esquema abaixo lê três sensores analógicos. Cada um corresponde a um valor de nota em uma série de notas. Se qualquer um dos sensores está acima de um determinado limiar, a nota correspondente será reproduzido.

Aqui está o esboço principal:

```
/*
 Teclado

  Reproduz uma nota que muda com base na mudança de uma entrada analógica

  circuito:
```

```
  * 3 resistores sensíveis à força de +5 V para
analógico em 0 a 5
  * 3 resistores de 10K analógica em 0 a 5 para
chão
  * Alto-falante de 8 ohms no pino digital 8

  criado 21 de janeiro de 2010
  modificado 09 de abril de 2012
  por Tom Igoe

Este exemplo de código é de domínio público.

 http://arduino.cc/en/Tutorial/Tone3

 */

#include "pitches.h"

const int threshold = 10;    // leitura mínima do
ensor que gerará a nota

// notas para tocar, correspondedo aos 3
sensosres:
int notes[] = {
  NOTE_A4, NOTE_B4,NOTE_C3 };

void setup() {

}

void loop() {
  for (int thisSensor = 0; thisSensor < 3; thisSensor
++) {
    // adquiri a leitura do sensor:
    int sensorReading = analogRead(thisSensor);

    // se o sensor é pressionado com força
suficiente:
    if (sensorReading > threshold) {
      // Toca a nota correspondente a este sensor:
      tone(8, notes[thisSensor], 20);
    }
  }
}
```

O esquema usa um arquivo extra, pitches.h. Este arquivo contém todos os valores de notas para notas típicas. Por exemplo, NOTE_C4 é meio C. NOTE_FS4 é fá sustenido, e assim por diante. Esta tabela de notas foi originalmente escrita por Brett Hagman, em cuja obra o comando *tone()* foi baseado. Você pode achar que é útil para quando você quiser fazer notas musicais.
Para fazer este arquivo, clique no botão "novo Tab" no canto superior direito da janela. Parece com este:

A cópia no seguinte código:

```
/*************************************************
 * Public Constants
 *************************************************/

#define NOTE_B0  31
#define NOTE_C1  33
#define NOTE_CS1 35
#define NOTE_D1  37
#define NOTE_DS1 39
#define NOTE_E1  41
#define NOTE_F1  44
#define NOTE_FS1 46
#define NOTE_G1  49
#define NOTE_GS1 52
#define NOTE_A1  55
#define NOTE_AS1 58
#define NOTE_B1  62
#define NOTE_C2  65
#define NOTE_CS2 69
#define NOTE_D2  73
#define NOTE_DS2 78
#define NOTE_E2  82
#define NOTE_F2  87
#define NOTE_FS2 93
#define NOTE_G2  98
#define NOTE_GS2 104
#define NOTE_A2  110
#define NOTE_AS2 117
#define NOTE_B2  123
#define NOTE_C3  131
#define NOTE_CS3 139
#define NOTE_D3  147
#define NOTE_DS3 156
#define NOTE_E3  165
#define NOTE_F3  175
#define NOTE_FS3 185
#define NOTE_G3  196
#define NOTE_GS3 208
#define NOTE_A3  220
#define NOTE_AS3 233
#define NOTE_B3  247
#define NOTE_C4  262
#define NOTE_CS4 277
#define NOTE_D4  294
#define NOTE_DS4 311
#define NOTE_E4  330
#define NOTE_F4  349
#define NOTE_FS4 370
#define NOTE_G4  392
#define NOTE_GS4 415
#define NOTE_A4  440
#define NOTE_AS4 466
#define NOTE_B4  494
#define NOTE_C5  523
#define NOTE_CS5 554
#define NOTE_D5  587
#define NOTE_DS5 622
#define NOTE_E5  659
#define NOTE_F5  698
#define NOTE_FS5 740
#define NOTE_G5  784
#define NOTE_GS5 831
```

```
#define NOTE_A5  880
#define NOTE_AS5 932
#define NOTE_B5  988
#define NOTE_C6  1047
#define NOTE_CS6 1109
#define NOTE_D6  1175
#define NOTE_DS6 1245
#define NOTE_E6  1319
#define NOTE_F6  1397
#define NOTE_FS6 1480
#define NOTE_G6  1568
#define NOTE_GS6 1661
#define NOTE_A6  1760
#define NOTE_AS6 1865
#define NOTE_B6  1976
#define NOTE_C7  2093
#define NOTE_CS7 2217
#define NOTE_D7  2349
#define NOTE_DS7 2489
#define NOTE_E7  2637
#define NOTE_F7  2794
#define NOTE_FS7 2960
#define NOTE_G7  3136
#define NOTE_GS7 3322
#define NOTE_A7  3520
#define NOTE_AS7 3729
#define NOTE_B7  3951
#define NOTE_C8  4186
#define NOTE_CS8 4435
#define NOTE_D8  4699
#define NOTE_DS8 4978
```

SPlaying tons em várias saídas usando a função *tone()*

Este exemplo mostra como usar o comando *tone()* para tocar notas diferentes em várias saídas.

O comando *tone()* funciona assumindo um dos timers internos do Atmega, definindo-o com a freqüência que você quer, e usando o timer para pulsar o pino de saída. Uma vez que é apenas usando um timer, você só pode tocar uma nota de cada vez. Você pode, no entanto, tocar notas em vários pinos em sequência. Para fazer isso, você precisa ativar o temporizador para um pino antes de passar para a próxima.

Graças ao Greg Borenstein por esclarecer isso.

Hardware Requerido

- (3) Auto-falantes de 8-ohm
- (3) Resistor de 100 ohm
- Placa de ensaio (breadboard)
- Fio de conexão

Circuito

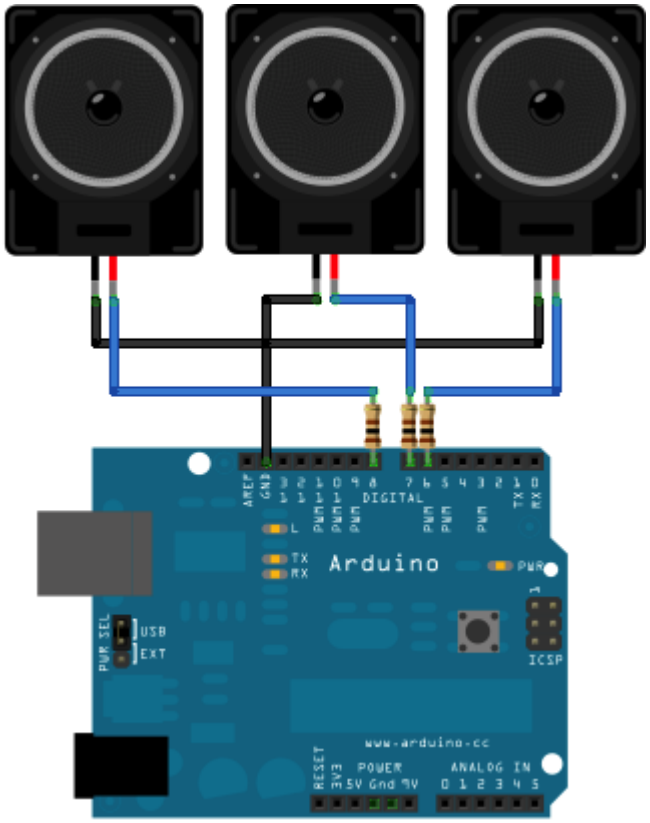
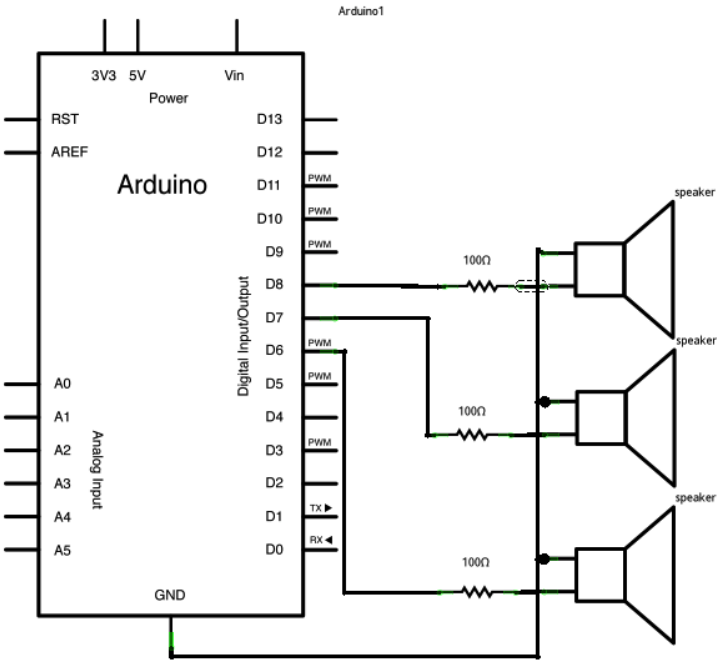


Imagem desenvolvida usando Fritzing. Para mais exemplos de circuitos, consulte a página do projeto Fritzing

Esquema



Código

O esquema abaixo desempenha um tom em cada um dos auto-falantes em sequência, desligando o auto-falante anterior primeiro. Note que a duração de cada tom é o mesmo que o atraso que se lhe segue.

Aqui está o esquema principal:

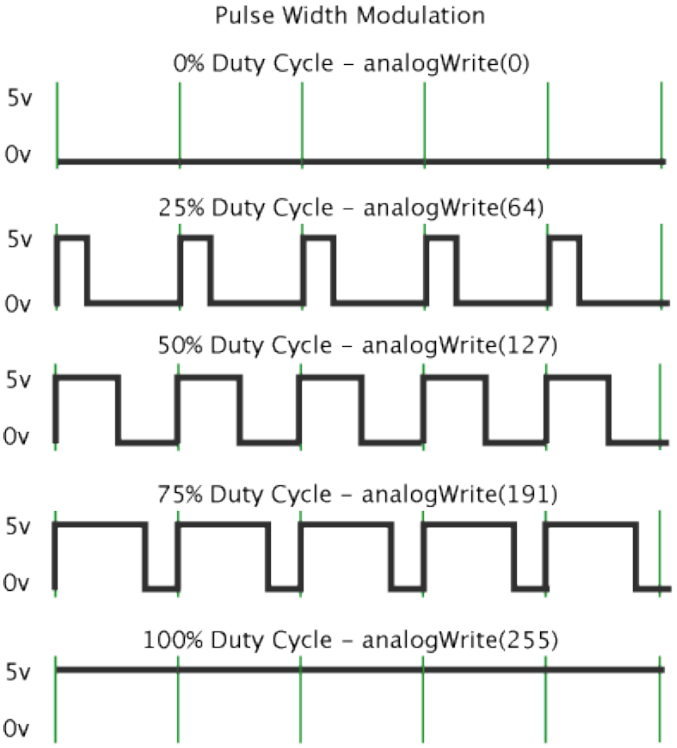
```
/*
Várias tocadores de tom

  Reproduz vários tons em vários pinos em sequência

  circuito:
  * 3 auto-falantes de 8 ohms nos pinos digitais 6, 7 e 8

  criado 08 de março de 2010
  por Tom Igoe
  com base em um trecho de Greg Borenstein

Este exemplo de código é de domínio público.

 http://arduino.cc/en/Tutorial/Tone4

 */

void setup() {

}

void loop() {
  // desliga a função tone no pino 8:
  noTone(8);
  // toca a nota no pino 6 por 200 ms:
  tone(6, 440, 200);
  delay(200);

  // desliga a função tone do pino 6:
  noTone(6);
  // toca a nota no pino 7 por 500 ms:
  tone(7, 494, 500);
  delay(500);

  // desliga a função tone no pino 7:
  noTone(7);
  // toca a nota no pino 8 por 500 ms:
  tone(8, 523, 300);
  delay(300);
}
```

PWM

O exemplo demonstra o uso de desvanecimento de saída analógica (PWM) para atenuar um LED. Isto está disponível no File->Sketchbook->Examples->Analog menu do software Arduino.

Modulação de largura de pulso, ou PWM, é uma técnica para a obtenção de resultados analógicos por meios digitais. O controle digital é usado para criar uma onda quadrada, um sinal alternado entre ligado e desligado. Este padrão on-off pode simular tensões entre totalmente ligado (5 Volts) e desligado (0 volts), alterando a parte do tempo que o sinal fica ligado em relação ao tempo que o sinal fica desligado. A duração de "no tempo" é chamado de largura de pulso. Para obter diferentes valores analógicos, você muda, ou modula, a largura do pulso. Se repetirmos este modelo de ligar-desligar o suficientemente rápido como um LED, por exemplo, o resultado é, como se o sinal fosse uma tensão constante entre 0 e 5v controlando a luminosidade do LED.

No gráfico abaixo, as linhas verdes representam um período de tempo regular. Esta duração ou período é o inverso da frequência do PWM. Em outras palavras, com freqüência PWM do Arduino em cerca de 500Hz, as linhas verdes medem 2 milissegundos cada. Uma chamada para *analogWrite()* em uma escala de 0-255, de tal modo que *analogWrite(255)* solicita um ciclo de trabalho de 100% (sempre ligado) e *analogWrite(127)* é um ciclo de trabalho de 50% (na metade do tempo) por exemplo .



Uma vez que consiga este exemplo em execução, pegue o seu arduino e agite-o para trás e para frente. O que você está fazendo aqui é essencialmente o mapeamento de tempo em todo o espaço. Aos nossos olhos, o movimento borra cada piscar do LED em uma linha. Como o LED acende-se ou apaga-se, aquelas pequenas linhas irão aumentar e diminuir de comprimento. Agora você está vendo a largura do pulso.

Escrito por Timothy Hirzel

Time

millis()

Descrição

Retorna o número de milissegundos desde que a placa Arduino começou a executar o programa atual. Este número transbordará (voltará a zero), depois de aproximadamente 50 dias.

Parâmetros

Nenhum

Respostas

Número de milissegundos desde que o programa iniciou (*unsigned long*)

Exemplo

```
unsigned long time;

void setup(){
  Serial.begin(9600);
}
void loop(){
  Serial.print("Time: ");
  time = millis();
  //mostra o tempo desde que o programa iniciou
  Serial.println(time);
  // espera um segundo para não enviar um número massivo de dados
  delay(1000);
}
```

Dica:

Note que o parâmetro para millis é um longo sem sinal (*unsigned long*), erros podem ser gerados se um programador tenta fazer matemática com outros tipos de dados, como inteiros.

micros()

Descrição

Retorna o número de microssegundos desde que a placa Arduino começou a executar o programa atual. Este número transbordará (voltará a zero), após cerca de 70 minutos. Em placas Arduino de 16 MHz (por exemplo, Arduino Duemilanove e Nano), esta função tem uma resolução de quatro microssegundos (ou seja, o valor retornado é sempre um múltiplo de quatro). Em placas Arduino de 8 MHz (por exemplo, a LilyPad), esta função tem uma resolução de oito microssegundos.

Nota: existem 1.000 microssegundos em um milésimo de segundo e 1.000.000 microssegundos em um segundo.

Parâmetros

Nenhum

Respostas

Número de microssegundos desde que o programa iniciou (*unsigned long*)

Exemplo

```
unsigned long time;

void setup(){
  Serial.begin(9600);
}
void loop(){
  Serial.print("Time: ");
  time = micros();
  //mostra o tempo desde que o programa iniciou
  Serial.println(time);
  // espera um segundo para não enviar um número massivo de dados
  delay(1000);
}
```

delay()

Descrição

Interrompe o programa em uma quantidade de tempo (em milisegundos) especificado como parâmetro. (Há 1.000 milissegundos em um segundo).

Sintaxe

delay(ms)

Parâmetros

ms: o número de milissegundos para pausar (*unsigned long*)

Respostas

Nenhum

Exemplo

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

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

void                                                    loop()
{
  digitalWrite(ledPin, HIGH);   // define o LED em ligado
   delay(1000);                 // espera um segundo
  digitalWrite(ledPin, LOW);    // define o LED desligado
   delay(1000);                 // espera um segundo
}
```

Advertências

Embora seja fácil criar um LED piscante com a função *delay()*, muitos esquemas usam pequenos atrasos para tarefas como mudar o debouncing (método para certificar que um botão foi apertado por exemplo), o uso de *delay()* em um esquema tem desvantagens significativas. Nenhuma outra leitura de

sensores, cálculos matemáticos, ou manipulação de pinos podem continuar durante a função de atraso, então de fato, isto traz mais outra atividade a um impasse. Para abordagens alternativas para controlar o tempo ver a função *millis()* e o esquema situado abaixo. Programadores mais experientes costumam evitar o uso de *delay()* para o sincronismo de eventos com mais de 10 milésimos de segundo a menos que o esquema Arduino seja muito simples.
Certas coisas não continuam enquanto a função *delay()* está controlando o chip Atmega, no entanto, porque a função de atraso não desabilita as interrupções. A comunicação serial que aparece no pino RX é registrada, valores PWM (*analogWrite*) e estados de pinos são mantidos, e as interrupções irão funcionar como deveriam.

delayMicroseconds()

Descrição

Interrompe o programa na quantidade de tempo (em microssegundos) especificada nos parâmetro. Há milhares de microssegundos em um milésimo de segundo, e um milhão de microssegundos em um segundo.
Atualmente, o maior valor que irá produzir um atraso preciso é 16383. Isso poderá mudar em futuras versões do Arduino. Para atrasos maiores do que alguns milhares de microssegundos, você deve usar o *delay()* em seu lugar.

Sintaxe

```
delayMicroseconds(us)
```

Parâmetros

us: o número de microssegundos da pausa (*unsigned int*)

Respostas

Nenhum

Exemplo

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

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

void loop()
{
  digitalWrite(outPin, HIGH);   // define o pino como ligado
  delayMicroseconds(50);        // pausa de 50 microsegundos
  digitalWrite(outPin, LOW);    // define o pino como desligado
  delayMicroseconds(50);        // pausa de 50 microsegundos
}
```

Configura o número do pino para trabalhar como pino de saída. Ele envia um trem de pulsos com período de 100 microssegundos.

Advertências e problemas conhecidos

Esta função funciona com muita precisão na faixa de 3 microssegundos para cima. Não podemos assegurar que *delayMicroseconds* funcionará precisamente em tempos de espera menores.
A partir do Arduino 0018, *delayMicroseconds()* não desabilita interrupções.

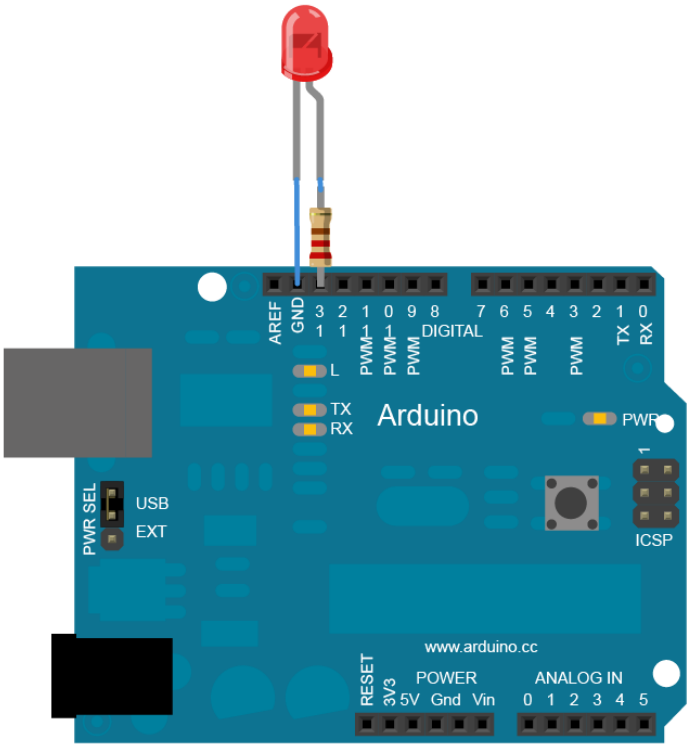
Blink Without Delay

Sometimes you need to do two things at once. For example you might want to blink an LED (or some other time-sensitive function) while reading a button press or other input. In this case, you can't use delay(), or you'd stop everything else the program while the LED blinked. The program might miss the button press if it happens during the delay(). This sketch demonstrates how to blink the LED without using delay(). It keeps track of the last time the Arduino turned the LED on or off. Then, each time through loop(), it checks if a long enough interval has passed. If it has, it toggles the LED on or off.
Às vezes você precisa fazer duas coisas ao mesmo tempo. Por exemplo, você pode querer piscar um LED (ou alguma outra função sensíveis ao tempo) durante a leitura do pressionamento de um botão ou outra entrada. Neste caso, você não pode usar *delay()*, ou irá parar todo o resto o programa enquanto o LED piscou. O programa pode perder o pressionamento do botão, se isso acontecer durante o *delay()*. Este esquema demonstra como a piscar o LED sem usar *delay()*. Ele mantém um registro da última vez que o Arduino tornou o LED ligado ou desligado. Então, cada vez que completar o *loop()*, ele verifica se passou durante um intervalo de tempo suficiente. Se ele tiver, ele alterna o LED entre ligado ou desligado.

Hardware Requerido

- Placa Arduino
- LED

Circuito



Para construir o circuito, pegue um LED e fixe a perna longa, perna positivo (chamado de anodo) ao pino 13. Fixe a perna curta, negativa (chamado de cátodo) ao terra. Em seguida, conecte sua placa Arduino em seu computador, inicie o programa Arduino, e insira o código abaixo.

Esquema:

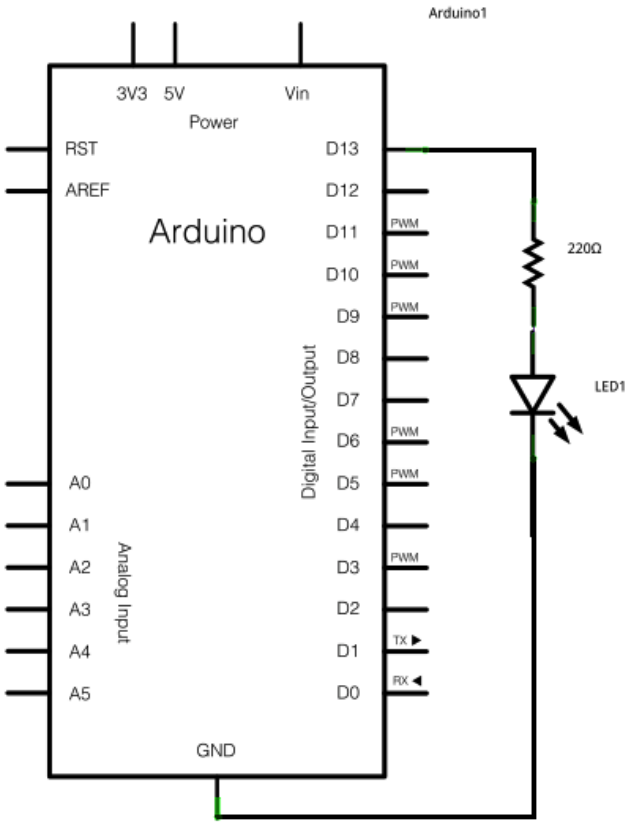


Imagem desenvolvida usando Fritzing. Para mais exemplos de circuitos, consulte a página do projeto Fritzing

O código a seguir usa a função *millis()*, um comando que retorna o número de milissegundos desde que a placa Arduino começou a rodar o programa atual, para piscar um LED.

```
/* Piscar sem Delay

 Liga e desliga um diodo emissor de luz (LED) conectado a um pino digital, sem usar a função delay(). Isto significa que outros códigos podem ser executados ao mesmo tempo sem serem interrompidos pelo código do LED.

 O circuito:
 * LED ligado do pino 13 ao terra.
 * Nota: na maioria dos Arduinos, já existe um LED na placa que está ligado ao pino 13, então não é necessário nenhum equipamento para este exemplo.


 criado 2005
 por David A. Mellis
 modificado 8 de fevereiro de 2010
 por Paul Stoffregen

 Este exemplo de código é de domínio público.

 http://www.arduino.cc/en/Tutorial/BlinkWithoutDelay
 */

// constantes não vão mudar. Usado aqui para
// definir o número dos pinos:
const int ledPin =  13;      // o número do pino do LED

// Variáveis irão mudar:
int ledState = LOW;          // ledState usado para definir o LED
long previousMillis = 0;        // irá armazenar a última vez que o LED foi atualizado

// as seguintes variáveis são longas por causa do temo, mesuradas em milissegundos,
// rapidamente irá começar um grande número que irá ser armazenada em um inteiro.
long interval = 1000;           // intervalo no qual vai piscar (milisegundos)

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

void loop()
{
  // aqui é onde você coloca o código que precisa ser executado durante todo o tempo.

  // verifique se é hora de piscar o LED, ou seja, se a
```

```
  // diferença entre o tempo atual e a última vez que você piscou
  // o LED é maior do que o intervalo no qual você deseja
  // Pisca o LED.
 unsigned long currentMillis = millis();

 if(currentMillis - previousMillis > interval) {
  // salva a última vez que você piscou o LED
  previousMillis = currentMillis;

  // se o LED estiver desligado liga-o e vice-versa:
  if (ledState == LOW)
    ledState = HIGH;
  else
    ledState = LOW;

  // define o LED com a varíavel de ledState:
  digitalWrite(ledPin, ledState);
 }
}
```

Math

min(x, y)

Descrição

Calcula o mínimo de dois números.

Parâmetros

x: o primeiro número, de qualquer tipo de dados
y: o segundo número, de qualquer tipo de dados

Respostas

O menor de dois números.

Exemplos

```
sensVal = min(sensVal, 100); // especifica sensVal no menor de sensVal de 100
                             // assegura que ele nunca esteja acima de 100.
```

Notas

Talvez contra-intuitiva, *max()* é muitas vezes usado para confinar a extremidade menor em um intervalo de variáveis, enquanto *min()* é usada para restringir a extremidade superior do intervalo.

Cuidados

Devido à maneira como a função *min()* é implementada, evitar o uso de outras funções dentro dos colchetes, pode levar a resultados incorretos
min (a++, 100) / / evitar isso - produz resultados incorretos
a++;
min(a, 100); // usar isso ao invés - conserva outro cálculos fora da função

max(x, y)

Descrição

Calcula o máximo de dois números.

Parâmetros

x: o primeiro número, de qualquer tipo de dados
y: o segundo número, de qualquer tipo de dados

Respostas

O maior dos valores de dois parâmetros.

Exemplo

```
sensVal = max(senVal, 20); // define sensVal como o maior sensVal de 20
                           // (garante efetivamente que é pelo menos 20)
```

Notas

Talvez contra-intuitiva, *max()* é muitas vezes usado para confinar a extremidade inferior de um intervalo de variáveis, enquanto *min()* é usada para restringir a extremidade superior do intervalo.

Cuidados

Devido à maneira como a função *max()* é implementada, evitar o uso de outras funções dentro dos colchetes, pode levar a resultados incorretos
max (a--, 0) / / evitar isso - produz resultados incorretos

a - / / usar isso ao invés - *max(a, 0)* // mantém outro cálculo fora da função

abs(x)

Descrição

Calcula o valor absoluto de um número.

Parâmetros

x: o número

Respostas

x: se x é maior ou igual a 0.
-x: se x é menor do que 0.

Cuidados

Devido à maneira como a função *abs()* é implementada, evitar o uso de outras funções dentro dos parênteses, pois pode levar a resultados incorretos.
abs (a + +) / / evitar isso - produz resultados incorretos

a + + / / usar isso ao invés -
abs (a) // mantém outro cálculo fora da função

constrain(x, a, b)

Descrição

Restringe um número para ficar num intervalo.

Parâmetros

x: o número restringido, todos os tipos de dados

a: o menor valor do intervalo, qualquer tipo de dados
b: o maior valor do intervalo, qualquer tipo de dados

Respostas

x: se x está entre a e b
a: se x é menor que a
b: se x for maior que b

Exemplo

```
sensVal = constrain(sensVal, 10, 150);
// limita o intervalo dos valores do sensor entre 10 e 150
```

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

Descrição

Re-mapeia um número a partir de um intervalo para outro. Ou seja, um valor de *fromLow* iria ficar mapeado para *toLow*, um valor de *fromHigh* para *toHigh*, valores entre o intervalo para valores entre o intervalo, etc
Não restrinja os valores dentro do intervalo, pois os valores de fora do intervalo são por vezes pretendido e úteis. A função de *constrain()* pode ser usada antes ou após esta função, se os limites para os intervalos forem desejados.
Note-se que os “limites inferiores” de qualquer dos intervalos pode ser maior ou menor do que os "limites superiores" de modo que a função *map()* pode ser usada para inverter uma série de números, por Exemplo
y = map(x, 1, 50, 50, 1);
A função também lida bem com números negativos, então aqui está o exemplo
y = map(x, 1, 50, 50, -100);
também é válido e funciona bem.
A função *map()* usa números inteiros, por isso não irá gerar frações, quando a matemática pode indicar que ele deve fazê-lo. Restos fracionários são truncados, e não são arredondados ou exibidos em média.

Parâmetros

value: o número de map
fromLow: o limite inferior do valor do intervalo corrente
fromHigh: o limite superior do valor do intervalo corrente
toLow: o limite inferior do valor da meta
toHigh: o limite superior do valor da meta

Respostas

O valor mapeado.

Exemplo

```
/* Map é um valor analógico de 8 bits (0 a 255) */
void setup() {}

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

Apêndice

Para os com inclinação pela matemática, 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;
}
```

pow(base, exponent)

Descrição

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

Parâmetros

base: o número (*float*)
exponent: a potência a qual a base é elevada (*float*)

Respostas

O resultado da exponenciação (*double*)

Exemplo

Veja a função *fscale* na biblioteca de códigos.

sqrt(x)

Descrição

Calcula a raiz quadrada de um número.

Parâmetros

x: o número, de qualquer tipo de dados

Respostas

double, a raiz quadrada do número

Trigonometria

sin(rad)

Descrição

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

Parâmetros

rad: o ângulo em radianos (*float*)

Respostas

O seno do ângulo (*double*)

cos(rad)

Descrição

Calcula o cosseno de um ângulo (em radianos). O resultado vai ser entre -1 e 1.

Parâmetros

rad: o ângulo em radianos (*float*)

Respostas

O cosseno do ângulo ("double")

tan(rad)

Descrição

Calcula a tangente de um ângulo (em radianos). O resultado será entre o infinito negativo e infinito positivo.

Parâmetros

rad: o ângulo em radianos (*float*)

Repostas

A tangente do ângulo (*double*)

Números Aleatórios

randomSeed(seed)

Descrição

randomSeed() inicializa o gerador de números pseudo-aleatórios, fazendo com que ela comece em um ponto arbitrário em sua seqüência aleatória. Esta sequência, enquanto muito longo, e aleatória, é sempre a mesma.

Se é importante para uma seqüência de valores gerados por *random()* diferir, em execuções subseqüentes de um esquema, utilize *randomSeed()* para inicializar o gerador de números aleatórios com uma entrada bastante aleatória, como *analogRead()* em um pino desconectado.

Por outro lado, pode ocasionalmente ser útil usar seqüências pseudo-aleatórias que se repetem exatamente. Isto pode ser conseguido ligando *randomSeed()* com um número fixo, antes de iniciar a sequência aleatória.

Parâmetros

long, int - passar um número para gerar a semente (aleatórios).

Respostas

Sem respostas

Exemplo

```
long randNumber;

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

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

  delay(50);
}
```

random()

Descrição

A função aleatória gera números pseudo-aleatórios.

Sintaxe

random(max)
random(min, max)

Parâmetros

min - limite inferior do valor aleatório, inclusive (opcional)

max - limite superior do valor aleatório, exclusivo

Respostas

Um número aleatório entre o mínimo e o máximo -1 (*long*)

Nota:

Se é importante para uma seqüência de valores gerados por *random()* diferir, em execuções subseqüentes de um esquema, utilize *randomSeed()* para inicializar o gerador de números aleatórios com uma entrada bastante aleatória, como *analogRead()* em um pino desconectado.

Por outro lado, ele pode ocasionalmente ser útil usar seqüências pseudo-aleatórias que se repetem exatamente. Isto pode ser conseguido ligando *randomSeed()* com um número fixo, antes de iniciar a sequência aleatória.

Exemplo

```
long randNumber;

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

  //se a entrada analógica do pino 0 estiver desconectada, random analog
  // ruído poderá causar a chamada do randomSeed() para gerar
  // diferentes números aleatórios (sementes) a cada momento durante a execução do esquema.
  // randomSeed()Em seguida, mistura a função aleatória.
  randomSeed(analogRead(0));
}

void loop() {
  // print a random number from 0 to 299
  randNumber = random(300);
  Serial.println(randNumber);

  // print a random number from 10 to 19
  randNumber = random(10, 20);
  Serial.println(randNumber);
```

```
  delay(50);
}10
```

Bits e Bytes

lowByte()

Descrição

Extrai o byte de menor ordem (mais à direita) de uma variável (por exemplo, uma palavra).

Sintaxe

lowByte(x)

Parâmetros

x: um valor de qualquer tipo

Respostas

byte

highByte()

Descrição

Extrai o byte de maior ordem (mais à esquerda) de uma palavra (ou o segundo byte mais baixo de um tipo de dados maior).

Sintaxe

highByte(x)

Parâmetros

x: um valor de qualquer tipo

Respostas

byte

bitRead()

Descrição

Lê o bit de um número.

Sintaxe

bitRead(x, n)

Parâmetros

x: o número que será lido

n: o bit a ser lido, a partir de 0 para o bit menos significativo (mais à direita)

Respostas

O valor do bit (0 or 1).

bitWrite()

Descrição

Escreve um bit de uma variável numérica.

Sintaxe

bitWrite(x, n, b)

Parâmetros

x: a variável numérica para escrever

n: qual o bit do número a ser escrito, a partir de 0 para o bit menos significativo (mais à direita)

b: o valor para escrever o bit (0 or 1)

Respostas

Nenhuma

bitSet()

Descrição

Define (escreve o 1 para) o bit de uma variável numérica.

Sintaxe

bitSet(x, n)

Parâmetros

x: a variável numérica cujo o bit será definido

n: qual o bit a ser definido, a partir de 0 para o bit menos significativo (mais à direita)

Respostas

Nenhuma

bitClear()

Descrição

Limpa (escreve o 0 para) o bit de uma variável numérica.

Sintaxe

bitClear(x, n)

Parâmetros

x: a variável numérica cujo o bit será limpo

n: qual o bit a limpar, a partir de 0 para o bit menos significativo (mais à direita)

Respostas

Nenhuma

bit()

Descrição

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

Sintaxe

bit(n)

Parâmetros

n: o bit cujo valor é para calcular

Respostas

O valor do bit

Interrupções Externas

attachInterrupt()

Descrição

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 Arduino tem duas interrupções externas: Números 0 (no pino digital 2) e 1 (no pino digital 3). A tabela abaixo mostra os pinos de interrupção disponíveis em várias placas.

Board	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)					

A placa Arduino Due tem capacidades de interrupções poderosas que permite que você anexe uma função de interrupção em todos os pinos disponíveis. Você pode especificar diretamente o número de pinos em *attachInterrupt()*.

Nota

Dentro da função conectada, *delay()* não vai funcionar e o valor retornado pelo *millis()* não será incrementado. Dados seriais recebidos enquanto em função poderão ser perdidos. Você deve declarar como volátil (*volatile*) quaisquer variáveis que modifiquem dentro da função anexada. Veja a seção sobre ISRs abaixo para mais informações.

Usando Interrupções

Interrupções são úteis para fazer as coisas acontecerem automaticamente em programas microcontrolados, e pode ajudar a resolver problemas de tempo. Boas missões para usar uma interrupção podem incluir a leitura de um encoder rotativo, ou monitoramento de entrada do usuário.

Se você quiser garantir que um programa sempre pegue os pulsos de um encoder rotativo, para que ele nunca perca um pulso, seria muito difícil de escrever um programa para fazer qualquer coisa como isso, porque o programa teria que consultar constantemente o as linhas do sensor para o codificador, a fim de pegar os pulsos quando eles ocorreram. Outros sensores têm uma interface dinâmica semelhante também, como tentar ler um sensor de som que está tentando pegar um clique, ou um sensor de ranhura infravermelho (foto-interruptor) tentando pegar uma gota caída. Em todas estas situações, utilizando uma interrupção pode libertar o microcontrolador de algum outro trabalho enquanto não perde a entrada.

Sobre Rotinas de Interrupção de Serviços

ISRs são tipos especiais de funções que têm algumas limitações únicas que a maioria das outras funções não têm. Um ISR não pode ter qualquer parâmetro, e não deve retornar nada.

Geralmente, uma ISR deve ser a mais curta e rápida quanto possível. Se o seu esquema usa vários ISRs, apenas um pode ser executado ao mesmo tempo, outras interrupções serão ignoradas (desligadas) até que a atual seja concluída. Como *delay()* e *millis()* ambos dependem de interrupções, eles não irão funcionar enquanto um ISR estiver em execução. *delayMicroseconds()*, que não depende de interrupções, vai funcionar como o esperado.

Normalmente as variáveis globais são usadas para transmitir dados entre um ISR e o programa principal. Para certificar-se de que as variáveis usadas em um ISR estão atualizados corretamente, declare-as como volátil (*volatile*). Para mais informações sobre as interrupções, consulte as notas de Nick Gammon.

Sintaxe

attachInterrupt(interrupt, ISR, mode)

attachInterrupt(pin, ISR, mode) (*Arduino Due apenas*)

Parâmetros

interrupt: O número da interrupção (*int*)

pin: O número do pino (*Arduino Due apenas*)

ISR: o ISR para ser chamado quando ocorre a interrupção; esta função não deve tomar nenhum parâmetros e não retornar nada. Esta função é por vezes referida como rotina do serviço não interrompida.

mode: define quando a interrupção deve ser acionada. Quatro constantes são predefinidas como valores válidos:
• *LOW* para provocar a interrupção sempre que o pino for *LOW*,
• *CHANGE* para provocar a interrupção sempre que o pino modificar o valor
• *RISING* para disparar quando o pino vai de *LOW* para *HIGH*,
• *FALLING* para quando o pino vai de *HIGH* para *LOW*.
A placa Due permite também:
• *HIGH* para desencadear a interrupção sempre que o pino for *HIGH* (Arduino Due apenas).

Respostas
Nenhuma

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;
}
```

detachInterrupt()

Descrição
Desliga a interrupção dada.

Sintaxe
detachInterrupt(*interrupt*)
detachInterrupt(*pin*) (*Arduino Due apenas*)

Parâmetros
- interrupt: o número da interrupção a ser desativada (ver *attachInterrupt()* para mais detalhes).
- pin: o número do pino da interrupção a ser desativada (somente Arduino Due)

Interrupções

interrupts()

Descrição
Re-habilita as interrupções (depois de terem sido desativadas por *noInterrupts()*). Interrupções permitem que certas tarefas importantes aconteçam em segundo plano e são ativadas por padrão. Algumas funções não funcionarão enquanto as interrupções estam desativadas, e a comunicação de entrada pode ser ignorada. Interrupções podem atrapalhar um pouco no tempo do código, no entanto, pode ser desativadas para grupo particularmente críticos do código.

Parâmetros
Nenhum

Respostas
Nenhuma

Exemplo

```
void setup() {}

void loop()
{
  noInterrupts();
  // crítico, código tempo-sensitivo aqui
  interrupts();
  // outro código aqui
}
```

noInterrupts()

Descrição
Desabilita interrupções (você pode reativá-las com *interrupts()*). Interrupções permitem que certas tarefas importantes aconteçam em segundo plano e são ativadas por padrão. Algumas funções não funcionarão enquanto as interrupções estam desativadas, e a comunicação de entrada pode ser ignorada. Interrupções podem atrapalhar um pouco o tempo do código, no entanto, e pode ser desativado para grupos particularmente críticos do código.

Parâmetros
Nenhum.

Respostas
Nenhuma.

Exemplo

```
void setup() {}

void loop()
```

```
{
  noInterrupts();
  // crítico, código tempo-sensitivo aqui
  interrupts();
  // outro código aqui
}
```

Comunicação

Serial

Usado para comunicação entre a placa Arduino e um computador ou outros dispositivos. Todas as placas Arduino têm pelo menos uma porta serial (também conhecida como um UART ou USART): Serial. Comunica-se sobre os pinos digitais 0 (RX) e 1 (TX), bem como com o computador por USB. Assim, se você usar essas funções, você não poderá usar também os pinos 0 e 1 para entrada ou saída digital.

Você pode usar o monitor serial incorporado ao ambiente Arduino para se comunicar com uma placa Arduino. Clique no botão monitor serial na barra de ferramentas e selecione a mesma taxa de transmissão usado 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 se comunicar com o seu computador pessoal, você vai precisar de um adaptador adicional USB-para-serial, como eles não estão conectados à placa do mega USB-para-serial. Para usá-los para se comunicar com um dispositivo serial TTL externo, conecte o pino TX no pino RX do seu aparelho, o RX para o pino TX do seu dispositivo, e o terra (ground) do seu mega ao terra do seu dispositivo. (Não conecte esses pinos diretamente a uma porta serial RS232; eles operam em +/- 12V e poderá danificar sua placa Arduino).

O Arduino Due tem três portas seriais adicionais TTL de 3.3V: Serial1 nos pinos 19 (RX) e 18 (TX); Serial2 nos pinos 17 (RX) e 16 (TX), Serial3 nos pinos 15 (RX) e 14 (TX). Os pinos 0 e 1 também são ligados aos pinos correspondentes do chip Serial ATmega16U2 USB-para-TTL, o qual está ligado à porta de depuração USB. Além disso, há uma porta serial USB nativa no chip SAM3X, SerialUSB.

A placa Arduino Leonardo usa Serial1 para se comunicar via TTL serial (5V) nos pinos 0 (RX) e 1 (TX). Serial é reservado para a comunicação USB CDC. Para mais informações, consulte a página de começo e página de hardware do Leonardo.

if (Serial)

Descrição

Indica se a porta serial especificada está pronta.

No Leonardo, *if* (Serial) indica quando ou não a conexão USB serial CDC está aberta. Para todos os outros casos, incluindo *if* (Serial1) no o Leonardo, este sempre retornará *true*.

Isto foi introduzido no Arduino 1.0.1.

Sintaxe

Todas as placas:
if (Serial)
Arduino Leonardo específico:
if (Serial1)
Arduino Mega específico:
if (Serial1)
if (Serial2)
if (Serial3)

Parâmetros

Nenhum

Respostas

Booleano: retorna true (verdadeiro) se a porta serial especificada estiver disponível. Isso só irá retornar falso se consultando a conexão serial USB CDC do Leonardo antes de estar pronto.

Exemplo:

```
void setup() {
  /Inicializar a serial e espera que a porta abra:
  Serial.begin(9600);
  while (!Serial) {
    ; // espera a porta serial conectar; Necessário apenas para Leonardo
  }
}

void loop() {
  //procede normalmente
}
```

available()

Descrição

Obtém o número de bytes (caracteres) disponíveis para leitura da porta serial. Este é um dado que já chegou e está armazenado no buffer de recepção serial (que detém 64 bytes). *available()* herda da classe utilitário Stream.

Sintaxe

Serial.available()

Arduino Mega apenas:
Serial1.available()
Serial2.available()
Serial3.available()

Parâmetros

Nenhum

Respostas

O número de bytes disponíveis para serem lidos

Exemplo

```
int incomingByte = 0;   // para dados serial recebidos

void setup() {
      Serial.begin(9600);   // abre a porta serial, define a taxa de dados em 9600 bps
}

void loop() {

    // envia dados apenas enquanto você recebe dados:
                if (Serial.available() > 0) {
                        // lê o byte recebido:
                    incomingByte = Serial.read();

                    // diz o que você recebeu:
                        Serial.print("I received: ");
                    Serial.println(incomingByte, DEC);
                                                }
}
```

Exemplo no Arduino Mega:

```
void setup() {
                                Serial.begin(9600);
                                Serial1.begin(9600);

}

void loop() {
    // Lê da porta 0, envia para a porta 1:
                if (Serial.available()) {
                  int inByte = Serial.read();
                    Serial1.print(inByte, BYTE);

                                                }
    // Lê da porta 1, envia para a porta 0:
                if (Serial1.available()) {
                  int inByte = Serial1.read();
                    Serial.print(inByte, BYTE);
                                                }
}
```

begin()

Descrição

Define a taxa de dados em bits por segundo (de transmissão) para transmissão de dados serial. Para se comunicar com o computador, use uma dessas taxas: 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600 ou 115200. Você pode, no entanto, especificar outras taxas - por exemplo, para comunicar sobre os pinos 0 e 1 com um componente que requer uma taxa de transmissão particular.
Um segundo argumento opcional configura os dados, paridade e bits de parada. O padrão é 8 bits de dados, sem paridade, um bit de parada.

Sintaxe

Serial.begin(speed)
Serial.begin(speed, config)
Arduino Mega apenas:
Serial1.begin(speed)
Serial2.begin(speed)
Serial3.begin(speed)
Serial1.begin(speed, config)
Serial2.begin(speed, config)
Serial3.begin(speed, config)

Parâmetros

speed: em bits por segundo (de transmissão) - longa
config: define os dados, paridade e bits de parada. Os valores válidos são:

- SERIAL_5N1
- SERIAL_6N1
- SERIAL_7N1
- SERIAL_8N1 (por padrão)
- SERIAL_5N2
- SERIAL_6N2
- SERIAL_7N2
- SERIAL_8N2
- SERIAL_5E1
- SERIAL_6E1
- SERIAL_7E1
- SERIAL_8E1
- SERIAL_5E2
- SERIAL_6E2
- SERIAL_7E2
- SERIAL_8E2
- SERIAL_5O1
- SERIAL_6O1
- SERIAL_7O1
- SERIAL_8O1
- SERIAL_5O2
- SERIAL_6O2
- SERIAL_7O2
- SERIAL_8O2

Respostas

Nada

Exemplo:

```
void setup() {
   Serial.begin(9600); // abre a porta serial, define a taxa de dados em 9600 bps
}

void loop() {}
```

Arduino Mega exemplo:

```
// Arduino Mega usando as quatro das suas portas seriais
//       (Serial,      Serial1,      Serial2,      Serial3),
//       com       diferentes       taxas       de       transmissões:

void                                                    setup(){
                                                  Serial.begin(9600);
                                                 Serial1.begin(38400);
                                                 Serial2.begin(19200);
                                                  Serial3.begin(4800);

                    Serial.println("Hello          Computer");
                Serial1.println("Hello          Serial          1");
                Serial2.println("Hello          Serial          2");
                Serial3.println("Hello          Serial          3");
}

void loop() {}
```

end()

Descrição

Desativa comunicação serial, permitindo que os pinos RX e TX possam ser usados para a entrada e saída em geral. Para reativar a comunicação serial, chame *Serial.begin()*.

Sintaxe

Serial.end()

Arduino Mega apenas:

Serial1.end()
Serial2.end()
Serial3.end()

Parâmetros

Nenhum

Respostas

Nenhuma

Serial.find()

Descrição

Serial.find () lê os dados do buffer serial até a sequecia alvo de determinado comprimento for encontrada. A função retorna *true* se a sequencia alvo for encontrada, *false* se expirar o tempo.

Serial.flush() herda da classe utilitário Stream.

Sintaxe

Serial.find(*target*)

Parâmetros

target : a sequencia a procurar por (char)

Respostas

booleana

Serial.findUntil()

Descrição

Serial.findUntil() lê os dados do buffer serial até que uma seqüência alvo de determinado comprimento ou terminador de sequencia for encontrada.

A função retorna *true* se a sequencua alvo for encontrada, *falso* se ela expirar.

Serial.findUntil() herda da classe utilitário Stream.

Sintaxe

Serial.findUntil(*target*, *terminal*)

Parâmetros

target : a sequencia a procurar por (char)
terminal : o terminador da sequencia na procura (char)

Respostas

booleana

flush()

Descrição

Espera para a transmissão de dados serial de saída finalize. (Antes do Arduino 1.0, esta alternativa removia quaisquer dados serial de entrada no buffer).

flush() herda da classe utilitário Stream.

Sintaxe

Serial.flush()

Arduino Mega apenas:

Serial1.flush()
Serial2.flush()
Serial3.flush()

Parâmetros

Nenhum

Respostas

Nada

Serial.parseFloat()

Descrição

Serial.parseFloat() retorna o primeiro número de ponto flutuante válido a partir do buffer serial. Caracteres que não são dígitos (ou o sinal de menos) são ignorados. *parseFloat()* é encerrado pelo primeiro caractere que não é um número de ponto flutuante.

Serial.parseFloat() herda da classe utilitário Stream.

Sintaxe

Serial.parseFloat()

Parâmetros

Nenhum

Respostas

float

parseInt()

Descrição

Procura o próximo inteiro válido no fluxo serial de entrada. *parseInt()* herda da classe utilitário Stream.

Se nenhum número inteiro válido é encontrado dentro de um segundo (ajustável através *Serial.setTimeout()*) um valor padrão de 0 será retornado.

Sintaxe

Serial.parseInt()

Arduino Mega apenas:

Serial1.parseInt()
Serial2.parseInt()
Serial3.parseInt()

Parâmetros

Nenhum

Respostas

int : o próximo inteiro válido

peek()

Descrição

Retorna o próximo byte (caractere) de dados serial de entrada, sem removê-lo do buffer serial interno. Ou seja, chamadas sucessivas para *peek()* irão retornar o mesmo caractere, como a próxima chamada para *read()*. *peek()* herda da classe de utilidades Stream.

Sintaxe

Serial.peek()

Arduino Mega apenas:

Serial1.peek()
Serial2.peek()
Serial3.peek()

Parâmetros

Nenhum

Respostas

O primeiro byte dos dados serial de entrada disponíveis (ou -1 se nenhum dado estiver disponível) - *int*

print()

Descrição

Exibe (print) os dados da porta serial como texto ASCII legível. Este comando pode assumir muitas formas. Os números são exibidos usando um caractere ASCII para cada dígito. *Floats* são igualmente exibidos como dígitos ASCII, padronizado com duas casas decimais. Bytes são enviados como um simples caractere. Caracteres e sequencias são enviados como ele. Por Exemplo:

• Serial.print (78) dá "78"
• Serial.print (1.23456) dá " 1.23"
• Serial.print ('N') dá "N"
• Serial.print ("Olá mundo") dá "Olá mundo".

Um segundo parâmetro opcional especifica a base (formato) para usar; valores permitidos são BIN (binário, ou de base 2), OCT (octal ou base 8), DEC (decimal , ou base 10) , HEX (hexadecimal , ou de base 16) . Para números de ponto flutuante, este parâmetro especifica o número de casas decimais a serem usadas. Por Exemplo:

• Serial.print (78, BIN) dá " 1001110 "
• Serial.print (78 de OCT) dá "116"
• Serial.print (78 de DEC) dá "78"
• Serial.print (78, HEX) dá " 4E "
• Serial.println (1.23456, 0) dá "1"
• Serial.println (1.23456, 2) dá " 1.23"
• Serial.println (1.23456, 4) dá " 1.2346 "

Você pode passar seqüências com base de memória flash para *Serial.print()* por envolvê-los com *F().* Por Exemplo:

• Serial.print (F ("Olá Mundo"))

Para enviar um único byte, use *Serial.write()*.

Sintaxe

Serial.print(val)
Serial.print(val, format)

Parâmetros

val: o valor a exibir – qualquer tipo de dados
format: especifica a base numérica (para tipo de dados integrais) ou número de casa decimais (para tipo de pontos flutuantes)

Respostas

size_t (long): *print()* retorna o número de bytes escritos, embora a leitura desses números é opcional

Exemplo:

```
/*
Use o ciclo FOR para dados e para exibir números em vários formatos
*/
int x = 0; // variável

void setup() {
  Serial.begin(9600); // abre a porta serial a 9600 bps:
}

void loop() {
  // exibe registros
  Serial.print("NO FORMAT"); // exibe a legenda
  Serial.print("\t"); // exibe a aba

  Serial.print("DEC");
  Serial.print("\t");

  Serial.print("HEX");
  Serial.print("\t");

  Serial.print("OCT");
  Serial.print("\t");

  Serial.print("BIN");
  Serial.print("\t");

  for(x=0; x< 64; x++){ // uma parte do quadro de ASCII,
```

```
muda a suíte

  // exibe-o em vários formatos:
  Serial.print(x);  // exibe o decodificador decimal ASCII – o mesmo para "DEC"
  Serial.print("\t");  // exibe a aba

  Serial.print(x, DEC);  // exibe o decodificador decimal ASCII
  Serial.print("\t");  // exibe a aba

  Serial.print(x, HEX);  // exibe o decodificador hexadecimal ASCI
  Serial.print("\t");  // exibe a aba

  Serial.print(x, OCT);  // exibe o decodificador octagonal ASCII
  Serial.print("\t");  // exibe a aba

  Serial.println(x, BIN);  // exibe o decodificador binário ASCII
  // então adiciona o símbolo de retorno com "println"
  delay(200);  // atraso de 200 milisegundos
 }
 Serial.println("");  // exibe outro símbolo de retorno
}
```

Dicas de programação

A partir da versão 1.0, a transmissão serial é assíncrona; *Serial.print()* irá retornar antes de quaisquer caracteres serem transmitidos.

println()

Descrição

Exibe os dados da porta serial como texto ASCII legível seguido por um caractere símbolo de retorno (ASCII 13, ou '\ r') e um caractere de nova linha (ASCII 10, ou '\ n'). Este comando assume as mesmas formas como *Serial.print().*

Sintaxe

Serial.println(val)
Serial.println(val, format)

Parâmetros

val: o valor a ser exibido – qualquer tipo de dados
format: especifica a base do número (para tipos de dados integrais) ou número de casas decimais (para tipos de ponto flutuantes).

Respostas

size_t (long): *println()* retorna o número de bytes escritos, embora a leitura desses números é opcional.

Exemplo:

```
/*
 entrada analógica

 lê uma entrada analógica em analógico em 0, exibe o valor para fora.

 Criada em 24 mar 2006
 por Tom Igoe
 */

int analogValue = 0;  // variável para armazenar o valor analógico

void setup() {
  // abre a porta serial a 9600 bps:
  Serial.begin(9600);
}

void loop() {
  // lê a porta analógica no pino 0:
  analogValue = analogRead(0);

  // exibi-o em vários formatos:
  Serial.println(analogValue);  // exibe codificado em ASCII decimal
  Serial.println(analogValue, DEC);  // exibe codificado em ASCII decimal
  Serial.println(analogValue, HEX);  exibe codificado em ASCII hexadecimal
  Serial.println(analogValue, OCT);  // exibe codificado em ASCII octal
  Serial.println(analogValue, BIN);  // exibe codificado em ASCII binary

  // espera 10 milisegundos antes da próxima leitura:
  delay(10);
}
```

read()

Descrição

Lê dados seriais de entrada. *read()* herda da classe utilitário Stream.

Sintaxe

Serial.read()
Arduino Mega apenas:
Serial1.read()
Serial2.read()
Serial3.read()

Parâmetros

Nenhum

Respostas

O primeiro byte disponível recebido dos dados serial (ou -1 se nenhum dado disponível) - *int*

Exemplo

```
int incomingByte = 0;  //para dados serial de entrada

void setup() {
  Serial.begin(9600);  // abre a porta serial, define a taxa de dados em 9600 bps
}

void loop() {

  // envia dados apenas quando recebe dados:
  if (Serial.available() > 0) {
    // lê o byte recebido:
```

```
                incomingByte = Serial.read();

                // diz o que você recebeu:
                Serial.print("I received: ");
                Serial.println(incomingByte, DEC);
                }
}
```

Serial.readBytes()

Descrição

Serial.readBytes() lê caracteres da porta serial em um buffer. A função termina se o comprimento determinado for lido, ou se o tempo acabar (ver *Serial.setTimeout()*).

Serial.readBytes() retorna o número de caracteres colocados no buffer. O 0 significa que nenhum dado válido foi encontrado.

Serial.readBytes() herda da classe utilitário Stream.

Sintaxe

Serial.readBytes(buffer, length)

Parâmetros

buffer: o buffer para armazenar o byte em (char[] ou byte[])

length : o número de bytes para ler (int)

Respostas

byte

Serial.readBytesUntil()

Descrição

Serial.readBytesUntil() lê os caracteres do buffer serial em uma matriz. A função termina se o caractere terminador é detectado, se comprimento determinado for lido, ou o tempo acabar (ver *Serial.setTimeout()*).

Serial.readBytesUntil() retorna o número de caracteres lidos no buffer. O 0 significa que nenhum dado válido foi encontrado.

Serial.readBytesUntil() herda da classe utilitário Stream.

Sintaxe

Serial.readBytesUntil(*character*, *buffer*, *length*)

Parâmetros

character : o caractere para procurar em (char)

buffer: o buffer que armazena os bytes em (char[] or byte[])

length : o número de bytes para ler (int)

Respostas

byte

Serial.setTimeout()

Descrição

Serial.setTimeout() define os milissegundos máximos de espera por dados serial ao usar *Serial.readBytesUntil()* ou *Serial.readBytes()*. O padrão é 1000 milissegundos.

Serial.setTimeout() herda da classe utilitário Stream

Sintaxe

Serial.setTimeout(*time*)

Parâmetros

time : tempo limite de duração em milisegundos (long).

Parâmetros

Nenhum

write()

Descrição

Grava dados binários para a porta serial. Estes dados são enviados como um byte ou uma série de bytes, para enviar os caracteres que representando os dígitos de um número, utilize a função *print()* em seu lugar.

Sintaxe

Serial.write(val)

Serial.write(str)

Serial.write(buf, len)

Arduino Mega também suporta: Serial1, Serial2, Serial3 (no lugar de Serial)

Parâmetros

val: o valor para ser enviada como um único byte

str: a sequencia para enviar como uma série de bytes

buf: uma matriz para enviar como uma série de bytes

len: o tamanho do buffer

Respostas

byte

write() irá retornar o número de bytes escrito, entretato a leitura do número é opcional

Exemplo

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

void                                    loop(){
   Serial.write(45); // envia o byte com o valor 45

  int bytesSent = Serial.write("hello"); //envia a sequencia
"hello" e retornar o comprimento da sequencia.
}
```

serialEvent()

Descrição

Chamado quando há dados disponíveis. Use *Serial.read()* para capturar esses dados.

PS: Atualmente, *serialEvent()* não é compatível com o Esplora, Leonardo, ou Micro

Sintaxe

```
void serialEvent(){
//declarações
}

Arduino Mega apenas:
void serialEvent1(){
//declarações
}
void serialEvent2(){
//declarações
}
void serialEvent3(){
//declarações
}
```

Parâmetros

statements: qualquer declaração válida

Stream

Stream é a classe base para o caractere e base de fluxos binários. Ele não é chamado diretamente, mas invocado sempre que você usar uma função que depende dele.

Stream define as funções de leitura no Arduino. Ao usar qualquer funcionalidade central que utiliza um *read()* ou método similar, você pode seguramente assumir que chama a classe Stream. Para funções como *print()*, Stream herda da classe de impressão.

Algumas das bibliotecas que dependem do fluxo incluem:

Biblioteca Wire (cabo)

Esta biblioteca permite que você se comunique com dispositivos I2C/TWI. Nas placas Arduino com o layout R3 (pinagem 1.0), a SDA (linha de dados) e SCL (linha de relógio) estão nos conectores dos pinos perto do pino AREF. O Arduino Due tem dois I2C/TWI interface SDA1 e SCL1 estão perto do pino AREF e um adicional está nos pinos 20 e 21.

Como uma referência a tabela abaixo mostra onde os pinos TWI estão localizados em várias placas Arduino.

Placas	I2C / TWI pinos
Uno, Ethernet	A4 (SDA), A5 (SCL)
Mega2560	20 (SDA), 21 (SCL)
Leonardo	2 (SDA), 3 (SCL)
Due	20 (SDA), 21 (SCL), SDA1, SCL1

A partir do Arduino 1.0, a biblioteca herda as funções Stream, tornando-a compatível com outras bibliotecas de leitura/gravação. Devido a isso, *send()* e *receive()* foram substituídos por *read()* e *write()*.

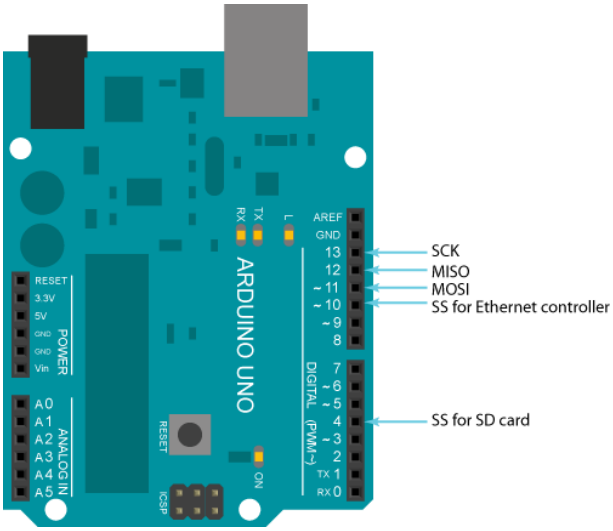
Notas

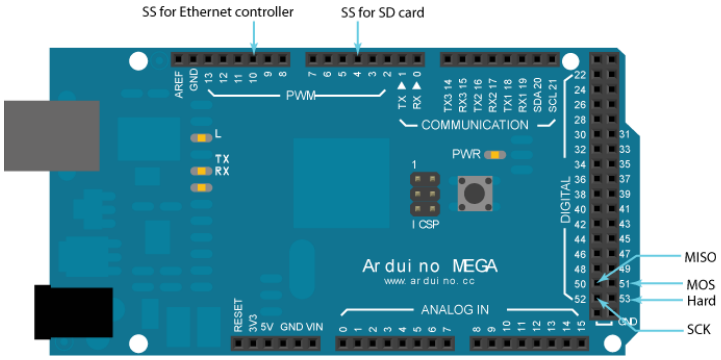
Há tanto versões de 7-bit e 8-bit de endereços I2C. 7 bits identificam o dispositivo, e o oitavo bit determina se ele está sendo escrito ou lido. A biblioteca Wire utiliza endereços 7 bits por toda parte. Se você tem um código de folha de dados ou de exemplo que usa o endereço de 8 bits, você vai querer largar o bit baixo (ou seja, mudar o valor um bit para a direita), produzindo um endereço entre 0 e 127.

Biblioteca Ethernet

Com o Arduino Ethernet Shield, esta biblioteca permite que uma placa Arduino se conecte à internet. Ela pode servir como um servidor e aceitar conexões de entrada ou um cliente fazendo as de saída. A biblioteca suporta até quatro conexões simultâneas (entrada ou saída ou uma combinação).

O Arduino se comunica com o shield usando o barramento SPI. Está nos pinos digitais 11, 12 e 13 no Uno e nos pinos 50, 51, e 52 no mega. Em ambas as placas, o pino 10 é usado como SS. No Mega, o pino de hardware SS, 53, não é usado para selecionar o W5100, mas deve ser mantido como uma saída ou a interface SPI não vai funcionar.





SD Library

A biblioteca SD permite a leitura e gravação em cartões SD, por exemplo, no Arduino Ethernet Shield. Ele é construído sobre *sdfatlib* por William Greiman. A biblioteca suporta sistema de arquivos FAT16 e FAT32 em sistemas de cartões SD standard e cartões SDHC. Ele usa nomes pequenos 8.3 para arquivos. Os nomes dos arquivos passados para as funções de biblioteca SD pode incluir pastas separadas por barras, /, por exemplo, "diretório/arquivo.txt". Porque o diretório de trabalho é sempre a raiz do cartão SD, um nome se refere ao mesmo arquivo ou não inclui uma barra inicial (por exemplo, "/file.txt" é equivalente a "file.txt"). A partir da versão 1.0, a biblioteca suporta abrir vários arquivos.

A comunicação entre o microcontrolador e o cartão SD usa SPI, que acontece nos pinos digitais 11, 12 e 13 (na maioria das placas Arduino) ou 50, 51 e 52 (Arduino Mega). Além disso, outro pino poderá ser usado para selecionar o cartão SD. Este pode ser o pino de hardware SS - pino 10 (na maioria das placas Arduino) ou pino 53 (no Mega) - ou outro pino especificado na chamada para *SD.begin()*. Note que mesmo se você não usar o pino de hardware SS, ele deve ser deixado como uma saída ou a biblioteca SD não vai funcionar.

Notas sobre usar a bibliotecar e vários shields

Wire.begin()
Wire.begin(address)

Descrição

Inicia a biblioteca Wire para fazer parte do barramento I2C como mestre ou escravo. Isso normalmente pode ser chamado apenas uma vez.

Parâmetros

address: o endereço de 7-bit escravo (opcional), se não for especificado, una o barramento como mestre.

Respostas

Nenhuma

Wire.requestFrom()

Descrição

Usado pelo mestre para solicitar bytes de um dispositivo escravo. Os bytes podem ser recuperados com as funções *available()* e *read()*.

A partir do Arduino 1.0.1, *requestFrom()* aceita um argumento booleano alterando seu comportamento para ser compatível com certos dispositivos I2C.

Se for *true*, *requestFrom()* envia uma mensagem de parada após a solicitação, liberando o barramento I2C.

Se *false*, *requestFrom()* envia uma mensagem de reinício após a solicitação. O barramento não será liberado, o que impede outro dispositivo mestre de pedir entre as mensagens. Isso permite que um dispositivo mestre envie várias solicitações, enquanto no controle.

O valor padrão é *true*.

Sintaxe

Wire.requestFrom(address, quantity)
Wire.requestFrom(address, quantity, stop)

Parâmetros

address: o endereço de 7-bit de um dispositivo para requisitar bytes de
quantity: o número de bytes requisitados
stop : booleano. *true* irá enviar a mensagem de parada após a requisição, liberando o barramento. *false* irá enviar mensagem de reinicio após a requisição, mantendo a conexão ativa.

Respostas

byte : o número de bytes retornados de um dispositivo escravo

Wire.beginTransmission(address)

Descrição

Começa uma transmissão para o dispositivo I2C escravo com o endereço fornecido. Posteriormente, forma fila de bytes para transmissão com a função *write()* e transmiti-os, chamando *endTransmission()*.

Parâmetros

address: o endereço de 7-bit do dispositivo para transmitir o

Respostas
Nenhuma

Wire.endTransmission()
Descrição
Termina a transmissão para um dispositivo escravo que foi iniciada por *beginTransmission()* e transmite os bytes que foram colocados em fila pelo *write()*.
A partir do Arduino 1.0.1, *endTransmission()* aceita um argumento booleano alterando seu comportamento para ser compatível com certos dispositivos I2C.
Se *true*, *endTransmission()* envia uma mensagem de parada após a transmissão, liberando o barramento I2C.
Se *false*, *endTransmission()* envia uma mensagem de reinício após a transmissão. O barramento não será liberado, o que impede que outro dispositivo mestre de transmitir entre as mensagens. Isso permite que um dispositivo mestre envie várias transmissões, enquanto no controle.
O valor padrão é *true*.

Sintaxe
Wire.endTransmission()
Wire.endTransmission(stop)
Parâmetros
stop : booleano. *true* irá enviar mensagem de parada, liberando o barramento após a transmissão. *false* irá enviar mensagem de reinicio, mantendo a conexão ativa.
Respostas
byte, que indica o estado da transmissão:

- 0: sucesso
- 1: dados muito grande para caber no buffer de transmissão
- 2: recebeu NACK em transmissão de endereço
- 3: recebeu NACK em transmissão de dados
- 4: outro erro

write()
Descrição
Grava dados de um dispositivo escravo em resposta a um pedido de um mestre, ou filas de bytes para a transmissão de um dispositivo mestre para o escravo (entre chamadas para *beginTransmission()* e *endTransmission ()*).

Sintaxe
Wire.write(value)
Wire.write(string)
Wire.write(data, length)
Parâmetros
value: o valor a enviar como um byte unitário
string: a sequencia a enviar como uma série de bytes
data: uma matriz de dados a enviar com bytes
length: o número de bytes a transmitir
Respostas
byte: *write()* irá retornar o número de bytes escritos, entretanto a leitura do número é opcional
Exemplo

```
#include <Wire.h>

byte val = 0;

void setup()
{
  Wire.begin(); // combina o barramento i2c
}

void loop()
{
  Wire.beginTransmission(44); // transmite para o dispositivo #44 (0x2c)
                              // o endereço do dispositivo é especificado na folha de dados
  Wire.write(val);            // envia o valor byte
  Wire.endTransmission();     // para a transmissão

  val++;                      // incrementa o valor
  if(val == 64) // se alcançar a posição 64 (max)
  {
    val = 0;    // recomeça do menor valor
  }
  delay(500);
}
```

Wire.available()
Descrição
Retorna o número de bytes disponíveis para recuperação com *read()*. Isto deve ser chamado pelo dispositivo mestre, após um chamado *torequestFrom()* ou em um escravo dentro do manipulador *onReceive()*.
available() herda da classe utilitário Stream.
Parâmetros
Nenhum
Respostas
O número de bytes disponível para leitura.

read()
Descrição
Lê o byte que foi transmitido a partir de um dispositivo escravo para um mestre após uma

chamada para *requestFrom()* ou foi transmitida a partir de um mestre para um escravo. *read()* herda da classe utilitário Stream.

Sintaxe

Wire.read()

Parâmetros

Nenhuma

Respostas

O próximo byte recebido

Exemplo

```
#include <Wire.h>

void setup()
{
  Wire.begin();      // combina o barramento i2c (endereço opcional para master)
    Serial.begin(9600);   // inicia o serial para saída
}

void loop()
{
    Wire.requestFrom(2, 6);      // solicita 6 bytes do dispositivo escravo #2

  while(Wire.available())   // o escravo talvez envie menos do que o solicitado
                                {
    char c = Wire.read();   // recebe o byte como caractere
        Serial.print(c);                // exibe o caractere
                                }

                                delay(500);
}
```

Wire.onReceive(handler)

Descrição

Registra a função a ser chamada enquanto o dispositivo escravo recebe a transmissão do mestre.

Parâmetros

handler: a função a ser chamada enquanto o escravo recebe dados; isto deve ter um único parâmetro int (o númer de bytes lido do mestre) e responde nada, exemplo:

```
void myHandler(int numBytes)
```

Respostas

Nenhuma

Wire.onRequest(handler)

Descrição

Registra a função a ser chamada quando o mestre solicita dados do dispositivo escravo.

Parâmetros

handler: a função a ser chamada, não tem nenhum parâmetro e não retorna nada, exemplo.:

```
void myHandler()
```

Respostas

Nenhuma

begin()

Descrição

Inicializa a biblioteca SD e o cartão. Isto começa o uso do barramento SPI (pinos digitais 11, 12 e 13 na maioria das placas Arduino, 50, 51 e 52 no mega) e o chip seleciona o pino, que por padrão é o pino de hardware SS (pino 10 na maioria das placas Arduino, 53 no mega). Observe que, mesmo se você usar um diferente chip que seleciona o pino, o pino de hardware SS deve ser mantido como uma saída ou as funções da biblioteca SD não irão funcionar.

Sintaxe

SD.begin()

SD.begin(cspin)

Parâmetros

cspin (*opcional*): o pino conectado a linha do chip que seleciona o cartão SD; padrões da linha de hardware SS do barramento SPI

Respostas

true no sucesso; *false* na falha

exists()

Descrição

Testa se o arquivo ou diretório existem no cartão SD.

Sintaxe

SD.exists(filename)

Parâmetros

filename: o nome do arquivo para testar a existência, que pode incluir diretórios (delimitados por barras, /)

Respostas

true se o diretório existe, *false* se não

mkdir()

Descrição

Crie um diretório no cartão SD. Isso também irá criar todos os diretórios intermediários que já não existem, por exemplo SD.mkdir ("a / b / c") vai criar a, b, e c.

Sintaxe

SD.mkdir(filename)

Parâmetros

filename: o nome do diretório para criar, com sub-diretórios separados por barra, /

Respostas

true se a criação do diretório for sucedida, *false* se não

open()

Descrição

Abre um arquivo no cartão SD. Se o arquivo é aberto para a escrita, ele será criado se ele não existir (mas o diretório que contém ele já deve existir).

Sintaxe

SD.open(filepath)
SD.open(filepath, mode)

Parâmetros

filename: o nome do arquivo a ser aberto, o que pode incluir diretórios (delimitados por barra, /) - Char *

Mode (opcional): o modo para abrir o arquivo, o padrão é file_read - byte. um de:

• FILE_READ: abrir o arquivo para leitura, a partir do início do arquivo.

• FILE_WRITE: abrir o arquivo para leitura e escrita, a partir do final do arquivo.

Respostas

Um objeto de arquivo referente ao arquivo aberto, se o arquivo não pôde ser aberto, este objeto irá avaliar como *false* em um contexto booleano, ou seja, você pode testar o valor de retorno com "*if(f)*".

remove()

Descrição

Remove o arquivo do cartão SD.

Sintaxe

SD.remove(filename)

Parâmetros

filename: o nome do arquivo a remover, o que pode incluir diretórios (delimitado por barra, /)

Respostas

true se a remoção do arquivo for realizada, *false* se não (se o arquivo não existir, o valor retornado não é especificado)

rmdir()

Descrição

Remove o diretório do cartão SD. O diretório deve estar vazio.

Sintaxe

SD.rmdir(filename)

Parâmetros

filename: o nome o diretório remover, com sub-diretórios separados por barra, /

Respostas

true se a remoção do diretório for realizada, *false* se não (se o diretório não existir, o valor retornado não é especificado)

available()

Checa se existem quaisquer bytes disponíveis para leitura do arquivo.
available() herda da classe utilitária Stream.

Sintaxe

file.available()

Parâmetros

file: a instância da classe de arquivos (retornado por *SD.open()*)

Respostas

O número de bytes disponíveis (*int*)

close()

Fecha o arquivo, e assegura que quaisquer dados escritos nele são salvos fisicamente no cartão SD.

Sintaxe

file.close()

Parâmetros

file: a instância da classe do arquivo (retornado por *SD.open()*)

Respostas

Nenhuma

flush()

Assegura que bytes escritos no arquivo são fisicamente salvos no cartão SD. Isto é realizado automaticamente enquanto o arquivo estiver fechado.
flush() herda da classe de utilidades Stream.

Sintaxe

file.flush()

Parâmetros

file: a instância da classe do arquivo (retornado por *SD.open()*)

Respostas

Nenhuma

peek()

Lê um byte do arquivo sem avançar para o próximo. Ou seja, chamadas sucessivas para *peek()* irão retornar o mesmo valor, como a próxima chamada para *read()*.
peek() herda da classe utilitário Stream.

Sintaxe

file.peek()

Parâmetros

file: a instância da classe de arquivos (retornado por *SD.open()*)

Respostas

O próximo byte (ou caractere), ou -1 se nenhum estiver disponível.

position()
Obtem a posição atual dentro do arquivo (ou seja, o local em que o próximo byte serão lidos ou gravados).
Sintaxe
file.position()
Parâmetros
file: a instância da classe de arquivos (retornado por *SD.open()*)
Respostas
A posição dentro do arquivo (unsigned long)

print()
Descrição
Exibe dados para o arquivo, que deve ter sido aberto para gravação. Exibe números como uma seqüência de dígitos, cada caractere ASCII (por exemplo, o número 123 é enviado como três caracteres '1', '2', '3 ').
Sintaxe
file.print(data)
file.print(data, BASE)
Parâmetros
file: a instância da classe de arquivos (retornado por *SD.open()*)
data: os dados a exibir (char, byte, int, long, or string)
BASE (opcional): a base em que será exibido os números: BIN para binário (base 2), DEC para decimal (base 10), OCT para octal (base 8), HEX para hexadecimal (base 16).
Respostas
byte
print() irá retornar o número de bytes escritos, entretanto a leitura dos números é opcional.

println()
Descrição
Exibe dados, seguido por um símbolo de retorno e nova linha, a do arquivo, que deve ter sido aberto para escrita. Exibe números como uma seqüência de dígitos, cada caractere ASCII (por exemplo, o número 123 é enviado como três caracteres '1', '2', '3').
Sintaxe
file.println()
file.println(data)
file.print(data, BASE)
Parâmetros
file: a instância da classe de arquivos (retornado por *SD.open()*)
data(opcional): os dados a exibir (char, byte, int, long, or string)
BASE (opcional): a base em que será exibido os números: BIN para binário (base 2), DEC para decimal (base 10), OCT para octal (base 8), HEX para hexadecimal (base 16).
Respostas
byte
println() irá retornar o número de bytes escritos, entretanto a leitura do número é opcional

seek()
Procura por uma nova posição no arquivo, que tem de estar entre 0 e o tamanho do arquivo (inclusive).
Sintaxe
file.seek(pos)
Parâmetros
file: a instância da classe de arquivos (retornado por *SD.open()*)
pos: a posição a procurar (*unsigned long*)
Respostas
true para sucesso, *false* para falha (*booleano*)

size()
Pega o tamanho do arquivo.
Sintaxe
file.size()
Parâmetros
file: a instância da classe de arquivos (retornado por *SD.open()*)
Respostas
O tamanho do arquivo e bytes (*unsigned long*)

read()
Lê o byte de um arquivo.
read() inerente da classe utilitária Stream.
Sintaxe
file.read()
Parâmetros
file: a instância da classe de arquivos (retornado por *SD.open()*)
Respostas
O próximo byte (ou caractere), ou -1 se nenhum estiver disponível.

write()
Descrição
Escreve dados no arquivo.
Sintaxe
file.write(data)
file.write(buf, len)
Parâmetros
file: a instância da classe de arquivos (retornado por *SD.open()*)
data: o byte, char, ou string (char*) a escrever

buf: uma sequencia de caracteres ou bytes
len: o número de elementos no buf

Respostas

byte

write() irá retornar o número de bytes escritos, entretanto a leitura dos números é opcional

isDirectory()

Diretórios (ou pastas) são um tipo especial de arquivos, a função reporta se o arquivo corrente é um diretório ou não.

Sintaxe

file.isDirectory()

Parâmetros

file: a instância da classe de arquivos (retornada por *file.open()*)

Respostas

booleano

Exemplo

```
#include <SD.h>

File root;

void setup()
{
  Serial.begin(9600);
  pinMode(10, OUTPUT);

  SD.begin(10);

  root = SD.open("/");

  printDirectory(root, 0);

  Serial.println("done!");
}

void loop()
{
  // nada acontece após a conclusão da configuração.
}

void printDirectory(File dir, int numTabs) {
  while(true) {

    File entry = dir.openNextFile();
    if (! entry) {
      // mais nenhum arquivo
      //Serial.println("**nomorefiles**");
      break;
    }
    for (uint8_t i=0; i<numTabs; i++) {
      Serial.print('\t');
    }
    Serial.print(entry.name());
    if (entry.isDirectory()) {
      Serial.println("/");
      printDirectory(entry, numTabs+1);
    } else {
      // arquivos tem tamanho, diretórios não tem
      Serial.print("\t\t");
      Serial.println(entry.size(), DEC);
    }
    }
}
```

openNextFile()

Relata o próximo arquivo ou pasta em um diretório.

Sintaxe

file.openNextFile()

Parâmetros

file: a instância da classe de arquivo que é um diretório

Respostas

char : o próximo arquivo ou pasta no caminho

Exemplo

```
#include <SD.h>

File root;

void setup()
{
  Serial.begin(9600);
  pinMode(10, OUTPUT);

  SD.begin(10);

  root = SD.open("/");

  printDirectory(root, 0);

  Serial.println("done!");
}

void loop()
{
  // nada acontece após a conclusão da configuração..
}

void printDirectory(File dir, int numTabs) {
  while(true) {

    File entry = dir.openNextFile();
    if (! entry) {
      // mais nenhum arquivo
      Serial.println("**nomorefiles**");
    }
    for (uint8_t i=0; i<numTabs; i++) {
      Serial.print('\t');
    }
    Serial.print(entry.name());
    if (entry.isDirectory()) {
      Serial.println("/");
      printDirectory(entry, numTabs+1);
    } else {
      // arquivos tem tamanho, diretórios não tem

      Serial.print("\t\t");
      Serial.println(entry.size(), DEC);
    }
```

```
                                                    }
}
```

rewindDirectory()

rewindDirectory() vai levar você de volta para o primeiro arquivo no diretório, usado em conjunto com *openNextFile()*.

Sintaxe

file.rewindDirectory()

Parâmetros

file: a instância da classe de arquivos.

Respostas

Nenhuma

Exemplo

```
#include                                    <SD.h>

File                                                root;

void                                                setup()
{
                                                Serial.begin(9600);
                                    pinMode(10,             OUTPUT);

                                                SD.begin(10);

                                root            =           SD.open("/");

                                    printDirectory(root,            0);

                                                Serial.println("done!");
}

void                                                loop()
{
  // nada acontece após a conclusão da configuração..
}

void    printDirectory(File     dir,    int     numTabs)    {
                                                while(true)             {

                                File    entry   =       dir.openNextFile();
                                                if      (!      entry)      {
                                            //  mais    nenhum  arquivo
                                    // retorna para o primeiro arquivo no diretório
                                                dir.rewindDirectory();
                                                break;
                                                }
                                for (uint8_t    i=0;    i<numTabs;  i++)    {
                                                Serial.print('\t');
                                                }
                                            Serial.print(entry.name());
                                        if      (entry.isDirectory())       {
                                                Serial.println("/");
                                    printDirectory(entry,   numTabs+1);
                                                }       else        {
    // arquivos tem tamanho, diretórios não tem

                                                Serial.print("\t\t");
                                    Serial.println(entry.size(),    DEC);
                                                }
                                                    }
}
```

STREAM

available()

Descrição

available() obtém o número de bytes disponíveis no fluxo. Isto é apenas para bytes que já chegaram.

Esta função é parte da classe Stream, e é chamado por qualquer classe que herda a partir dele (Wire, Serial, etc.) Veja a página principal classe de fluxo para mais informações.

Sintaxe

stream.available()

Parâmetros

stream : a instância da classe inerente a Stream.

Respostas

int : o número de bytes disponível para leitura

read()

Descrição

read() lê caracteres de um fluxo de entrada para o buffer.

Esta função é parte da classe Stream, e é chamado por qualquer classe que herda a partir dele (Wire, Serial, etc.) Veja a página principal classe de fluxo para mais informações.

Sintaxe

stream.read()

Parâmetros

stream: a instância da classe que é inerente a Stream.

Respostas

O primeiro byte dos dados recebidos disponível (ou -1 se nenhum dado estiver disponível)

flush()

Descrição

flush() limpa o buffer uma vez que todos os caracteres de saída tenham sido enviados.

Esta função é parte da classe Stream, e é chamado por qualquer classe que herda a partir dele (Wire, Serial, etc.) Veja a página principal classe de fluxo para mais informações.

Sintaxe

stream.flush()

Parâmetros

stream : a instância da classe que é inerente a Stream.

Respostas

booleano

find()

Descrição

find() lê os dados do fluxo até a sequencia alvo de determinado comprimento for encontrada. A função retorna *true* se a sequencia alvo for encontrada, *false* se o tempo esgotar.

Esta função é parte da classe Stream, e é chamado por qualquer classe que herda a partir dele (Wire, Serial, etc.) Veja a página principal classe de fluxo para mais informações.

Sintaxe

stream.find(*target*)

Parâmetros

stream : a instância da classe que é inerente a Stream.

target : a sequencia a procurar por (char)

Respostas

booleano

findUntil()

Descrição

findUntil() lê os dados do fluxo até a sequencia alvo de determinado comprimento ou terminador de sequencia ser encontrada.

A função retorna *true* se a sequencia alvo for encontrada, *false* se o tempo expirar

Esta função é parte da classe Stream, e é chamado por qualquer classe que herda a partir dele (Wire, Serial, etc.) Veja a página principal classe de fluxo para mais informações.

Sintaxe

stream.findUntil(*target*, *terminal*)

Parâmetros

stream : a instância da classe que é inerente a Stream.

target : a sequencia a procurar por (char)

terminal : o terminador da sequencia na procura (char)

Respostas

booleano

peek()

Lê o byte do arquivo sem avançar para o próximo. Ou seja, chamadas sucessivas para *peek()* irão retornar o mesmo valor, como a próxima chamada para *read()*.

Esta função é parte da classe Stream, e é chamado por qualquer classe que herda a partir dele (Wire, Serial, etc.) Veja a página principal classe de fluxo para mais informações.

Sintaxe

stream.peek()

Parâmetros

stream : a instância da classe que é inerente a Stream.

Respostas

O próximo byte (ou caractere), ou -1 se nenhum estiver disponível.

readBytes()

Descrição

readBytes() lê caracteres de um fluxo dentro do buffer. A função termina se o comprimento determinado for lido, ou se tempo terminou (ver *setTimeout()*).

readBytes() retorna o número de bytes colocados no buffer. O 0 significa que não foram encontrados dados válidos.

Esta função é parte da classe Stream, e é chamado por qualquer classe que herda a partir dele (Wire, Serial, etc.) Veja a página principal classe de fluxo para mais informações.

Sintaxe

stream.readBytes(buffer, length)

Parâmetros

stream : a instância da classe que é inerente a Stream.

buffer: o buffer que armazena os bytes em in (char[] ou byte[])

length : o número de bytes para ler (int)

Respostas

O número de bytes armazenados no buffer

readBytesUntil()

Descrição

readBytesUntil() lê caracteres de um fluxo em um buffer. A função termina se o caractere terminador for detectado, se o comprimento determinado for lido, ou se o tempo acabar (ver *setTimeout()*).

readBytesUntil() retorna o número de bytes colocados no buffer. O 0 significa que não foram encontrados dados válidos.

Esta função é parte da classe Stream, e é chamado por qualquer classe que herda a partir dele (Wire, Serial, etc.) Veja a página principal classe de fluxo para mais informações.

Sintaxe

stream.readBytesUntil(*character*, *buffer*, *length*)

Parâmetros

stream : a instância da classe que é inerente a Stream.

character : o caractere a procurar por (char)

buffer: o buffer que armazena bytes em (char[] ou byte[])

length : o número de bytes para ler (int)

Respostas
O número de bytes colocados no buffer

readString()
Descrição
readString() lê caracteres de um fluxo em uma sequencia. A função termina se o tempo expirar (ver *setTimeout()*).
Esta função é parte da classe Stream, e é chamado por qualquer classe que herda a partir dele (Wire, Serial, etc.) Veja a página principal classe de fluxo para mais informações.
Sintaxe
stream.readString()
Parâmetros
Nenhum
Respostas
A sequencia lida de um fluxo

readStringUntil()
Descrição
readStringUntil() lê caracteres de um fluxo de uma sequencia. A função termina se o caractere terminador é detectado ou o tempo acabar (ver *setTimeout()*).
Esta função é parte da classe Stream, e é chamado por qualquer classe que herda a partir dele (Wire, Serial, etc.) Veja a página principal classe de fluxo para mais informações.
Sintaxe
stream.readString(*terminator*)
Parâmetros
terminator : o caractere a procurar por (char)
Respostas
A sequencia inteira lida de um fluxo, até que seja detectado o caractere terminador.

parseInt()
Descrição
parseInt() retorna o primeiro número inteiro válido (long) a partir da posição atual. Caracteres iniciais que não são inteiros (ou o sinal de menos) são ignorados. *parseInt()* é encerrado pelo primeiro caractere que não é um dígito.
Esta função é parte da classe Stream, e é chamado por qualquer classe que herda a partir dele (Wire, Serial, etc.) Veja a página principal classe de fluxo para mais informações
Sintaxe
stream.parseInt(*list*)
Parâmetros
stream : a instância da classe que é inerente a Stream.
list : o fluxo a checar por ints (char)
Respostas
int

parseFloat()
Descrição
parseFloat() retorna o primeiro número de ponto flutuante válido a partir da posição atual. Caracteres iniciais que não são dígitos (ou o sinal de menos) são ignorados. *parseFloat()* é encerrado pelo primeiro caractere que não é um número de ponto flutuante.
Esta função é parte da classe Stream, e é chamado por qualquer classe que herda a partir dele (Wire, Serial, etc.) Veja a página principal classe de fluxo para mais informações.
Sintaxe
stream.parseFloat(*list*)
Parâmetros
stream : a instância da classe que é inerente de Stream.
list : o fluxo para checar flutuantes (char)
Respostas
float

setTimeout()
Descrição
setTimeout() define os milissegundos máximo de espera para o fluxo de dados, o padrão é 1000 milissegundos. Esta função é parte da classe Stream, e é chamado por qualquer classe que herda a partir dele (Wire, Serial, etc.) Veja a página principal classe Stream (fluxo) para mais informações.
Sintaxe
stream.setTimeout(*time*)
Parâmetros
stream : a instância da classe que é inerente a Stream.
time : a duração in milisegundos (long).
Parâmetros
Nenhum