

**UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ENGENHARIA DE LORENA**

TUTORIAL

**Experimentos e conceitos de programação computacional usando o
microcontrolador ARDUINO**

SUMÁRIO

Conteúdo	Pág.
Projeto 1: LED pisca pisca	1
Projeto 2: Código Morse	6
Projeto 3: Semáforo	9
Projeto 4: Semáforo interativo	10
Projeto 5: Perseguição de LEDs	12
Projeto 6: Efeito interativo de perseguição de LEDs	16
Projeto 7: Lâmpada pulsante	20
Projeto 8: Lâmpada de humor	22
Projeto 9: Efeito fogo de LED	21
Projeto 10: LEDs controlados	28
Projeto 11: Acionando um motor CC	35
Projeto 12: Atuador piezelétrico	38
Projeto 13: Sensor de temperatura com o CI LM35	31





Prof. Carlos Yujiro Shigue

**Lorena, SP
2014**

Projeto 1 - LED Pisca-Pisca

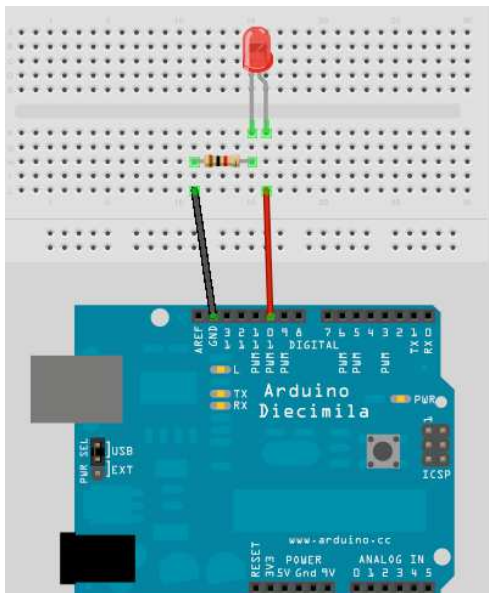
Neste projeto vamos configurar e programar o Arduino para fazer um LED piscar. Vamos usar LED vermelho para ensinar alguns conceitos de eletrônica e de codificação em C ao longo do caminho.

Componentes

Breadboard	
Red LED	
150Ω Resistor	
Jumper Wires	

Montagem

Primeiro, certifique-se que o seu Arduino está desligado. Você pode fazer isso desconectando o cabo USB ou tirando o jumper alimentador do Arduino. Em seguida, faça as ligações desta maneira:



Não importa se você usa diferentes fios coloridos ou se usa orifícios diferentes sobre a *proto*board, mas necessariamente os componentes e os fios têm de serem ligados da mesma forma que o mostrado na Figura acima. Tenha cuidado ao inserir os componentes na *proto*board. Se a

*proto*board for nova, o ajuste nos furos fará com que a inserção dos componentes seja um pouco mais dura. Deixar de inserir os componentes com cuidado pode resultar em danos. Verifique se o seu LED está ligado da maneira certa com a perna mais comprida conectada ao pino digital 10. A perna longa do LED é o anodo (+) e deve sempre ir para a alimentação de +5 V (neste caso, que sai do pino digital 10) e a perna mais curta é o catodo (-) e deve ir para o pino terra (GND). Quando você estiver certo de que tudo está ligado corretamente, conecte o Arduino ao computador pelo cabo USB.

A seguir, abra o IDE Arduino e digite o seguinte código:

```
// Projeto 1 - LED Pisca-Pisca
int ledPin = 10;
void setup() {
  pinMode(ledPin, OUTPUT);
}
void loop() {
  digitalWrite(ledPin, HIGH);
  delay(1000);
  digitalWrite(ledPin, LOW);
  delay(1000);
}
```

Agora pressione o Verificar / Compilar, botão no topo da IDE, para se certificar de que não há erros em seu código. Se esta for bem sucedida, agora você pode clicar no botão Upload para enviar o código para o Arduino. Se você tiver feito tudo certo, você deve ver agora o LED vermelho na *proto*board piscar e desligar a cada 1 segundo. Agora vamos dar uma olhada no código e no *hardware* e descobrir como ambos trabalham.

Descrição do código

```
// Projeto 1 - LED Pisca-Pisca
int ledPin = 10;
void setup() {
  pinMode(ledPin, OUTPUT);
}
void loop() {
  digitalWrite(ledPin, HIGH);
  delay(1000);
  digitalWrite(ledPin, LOW);
  delay(1000);
}
```

Então vamos dar uma olhada no código para este projeto, nossa primeira linha é:

```
// Projeto 1 - LED Pisca-Pisca
```

Isto é simplesmente um comentário no seu código e é ignorado pelo compilador (a parte do IDE que transforma o seu código em instruções do Arduino pode compreender antes de carregá-lo). Qualquer

texto inserido por trás de um comando `//` será ignorado pelo compilador e simplesmente está lá para você, ou qualquer outra pessoa que lê seus códigos. Comentários são essenciais em seu código para lhe ajudar a entender o que está acontecendo e como funciona o seu código. Os comentários também podem ser colocados após os comandos como na próxima linha do programa. Mais tarde, como os projetos se tornam mais complexos e seu código expande-se em centenas ou talvez milhares de linhas, os comentários serão vitais para torná-lo fácil para você ver como ele funciona. Você pode vir até com um espantoso pedaço de código, mas se você voltar e olhar esses códigos dias, semanas ou meses depois, você pode esquecer como tudo funciona. Os comentários irão ajudá-lo a compreendê-lo facilmente.

Além disso, se seu código é feito para ser visto por outras pessoas e como todo o ethos do Arduino, é de fato fonte aberta à comunidade, esperamos que quando você começar a fazer seu próprio material com o Arduino, você esteja disposto a compartilhá-lo com o mundo em seguida, comentários permitirão que as outras pessoas entendam o que está acontecendo no seu código. Você também pode colocar comentários em uma instrução usando em bloco comandos `/*` e `*/`.

Por exemplo, `/*` Todo o texto dentro da barra e os asteriscos é um comentário e será ignorado pelo compilador `*/` A IDE irá ligar automaticamente a cor de qualquer texto comentado com cinza.

A próxima linha do programa é:

```
int ledPin = 10;
```

Isto é conhecido como uma variável. Uma variável é um local para armazenar dados. Neste caso, você está configurando uma variável do tipo `int` ou inteiro. Um inteiro é um número na faixa de -32.768 a $+32.767$.

Em seguida, você atribuiu o nome inteiro de `ledPin` e deram-lhe um valor de 10. Nós não temos que chamá-lo de `ledPin`, poderíamos tê-lo chamado de qualquer coisa que quiséssemos. Mas, como queremos que o nosso nome da variável seja descritivo o chamamos de `ledPin` para mostrar que o uso desta variável é para definir quais os pinos no Arduino vamos usar para se conectar ao nosso LED. Neste caso, estamos usando o pino digital 10.

No final desta declaração, é um ponto e vírgula. Este é um símbolo para dizer ao compilador que esta declaração está completa. Apesar de podermos chamar as nossas variáveis do jeito que quisermos, cada nome de variável em C deve começar com uma letra, o resto do nome pode ser

composto de letras, números e caracteres de sublinhado. A linguagem C reconhece as letras maiúscula e minúscula como sendo diferentes.

E finalmente, você não pode usar as palavras-chave do C como `main`, para nomear as variáveis. Palavras-chave são constantes, variáveis e nomes de funções que são definidas como parte da linguagem do Arduino. Usar um nome variável é o mesmo que uma palavra-chave. Todas as palavras-chave no desenho irão aparecer em vermelho.

Assim, você irá configurar uma área na memória para armazenar um número do tipo inteiro e ter armazenado nessa área o número 10. Imagine uma variável como uma pequena caixa onde você pode manter as coisas. Uma variável é chamada de variável, porque você pode mudá-la. Mais tarde, vamos fazer os cálculos matemáticos sobre as variáveis para ampliar os nossos programas e fazer coisas mais avançadas.

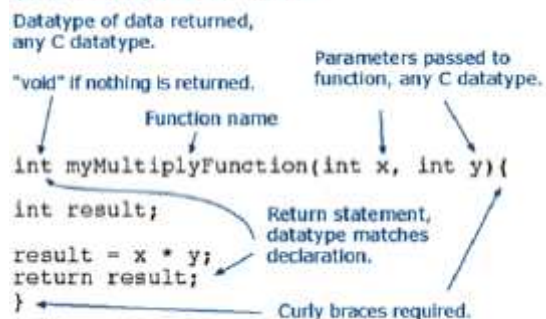
Em seguida, temos a nossa função `setup()`

```
void setup() {  
  pinMode(ledPin, OUTPUT);  
}
```

A função `setup()` é executada uma vez, e apenas uma vez no início do programa, onde você vai fazer emitir instruções gerais para preparar o programa antes das corridas `loop` principal, como a criação de modos de pinos, estabelecendo taxas de transmissão de série, etc.

Basicamente, uma função é um bloco de código montado em um bloco conveniente. Por exemplo, se nós criarmos a nossa própria função para levar a cabo uma série matemática complicada que tiveram muitas linhas de código, poderia executar esse código quantas vezes nós gostamos simplesmente chamando o nome da função em vez de escrevê-la novamente:

Anatomy of a C function



Mais tarde, vamos entrar em funções com mais detalhes quando começarmos a criar as nossas próprias manipulações. No caso do nosso programa o `setup()` funciona apenas como uma

instrução para realizar alguma tarefa. Outra função começa com:

```
void setup()
```

e aqui estamos dizendo ao compilador que a nossa função é chamada de configuração(`setup`), que não retorna dados (`void`) e que passamos sem parâmetros para ele (parênteses vazio).

```
int myFunc(int x, int y)
```

Neste caso, criamos uma função (ou um bloco de código) chamado `myFunc`. Esta função foi passada a dois inteiros chamados `x` e `y`. Uma vez que a função tem término irá retornar a um valor inteiro para o ponto depois de onde a nossa função foi chamada no programa (daí `int` antes do nome da função).

Todo o código dentro da função está contido dentro das chaves. Um símbolo `{` inicia o bloco de código e um símbolo `}` termina o bloco. Qualquer coisa entre estes dois símbolos é o código que pertence à função.

Nós vamos entrar em maiores detalhes sobre as funções mais tarde assim não se preocupe com eles agora. Tudo que você precisa saber é que neste programa temos duas funções: a primeira função é chamada de configuração (`setup`) e sua finalidade é dar a configuração necessária para o nosso programa para trabalhar antes que o laço principal do programa (`loop`) seja executado.

```
void setup() {  
  pinMode(ledPin, OUTPUT);  
}
```

Nossa função de instalação tem apenas uma declaração que é `pinMode`. Aqui nós estamos dizendo ao Arduino que queremos ajustar o modo de um dos nossos pinos digitais de saída para ser saída, em vez de entrada. Dentro do parêntese colocamos o número de pinos e o modo (`OUTPUT` ou `INPUT`). Nosso número de pinos é `ledPin`, que foi anteriormente definido para o valor 10 no nosso programa. Por conseguinte, esta declaração irá simplesmente dizer para o Arduino que o pino digital 10 deve ser configurado para o modo saída.

Como a função `setup()` é executada somente uma vez, passemos para o circuito principal:

```
void loop() {  
  digitalWrite(ledPin, HIGH);  
  delay(1000);  
  digitalWrite(ledPin, LOW);  
  delay(1000);  
}
```

O `void loop()` é a função principal do programa e é executado continuamente desde que o nosso Arduino esteja ligado. Toda declaração dentro da função `loop()` (dentro das chaves) é realizado, um por um, passo a passo, até que a parte inferior da função é atingida, e então o laço começa de novo no topo da função, e assim por diante para sempre ou até que você desligue o Arduino ou pressione o botão `reset`.

Neste projeto, queremos que o LED ligue, fique por um segundo, desligue e permaneça desligado por um segundo, e repita isso. Portanto, os comandos para dizerem ao Arduino para fazer o que está contido dentro do circuito (`void`) e funcionar como desejamos e repetir infinitamente.

A primeira declaração é:

```
digitalWrite(ledPin, HIGH);
```

e este escreve um sinal alto (`HIGH`) ou um valor baixo (`LOW`) para o pino digital na declaração (neste caso `ledPin`, que é o pino digital 10). Quando você definir um pino digital para `HIGH` você está mandando 5 volts para o pino. Quando você configurá-lo para `LOW` fica 0 volts.

Depois disso temos:

```
delay(1000);
```

Esta declaração simplesmente diz ao Arduino para esperar 1000 milissegundos (1 segundo) antes de realizar a próxima declaração de que é:

```
digitalWrite(ledPin, LOW);
```

que vai desligar a energia do pino digital 10 e portanto, desligar o LED. Há então outro atraso da declaração por mais 1000 milissegundos e depois a função termina. No entanto, como este é o nosso laço principal `loop()`, a função irá agora recomençar do início. Seguindo a estrutura do programa passo-a-passo, podemos ver que é muito simples.

```
// Projeto 1 - LED Pisca-Pisca  
int ledPin = 10;  
void setup() {  
  pinMode(ledPin, OUTPUT);  
}  
void loop() {  
  digitalWrite(ledPin, HIGH);  
  delay(1000);  
  digitalWrite(ledPin, LOW);  
  delay(1000);  
}
```

Começamos por atribuir uma variável chamada `ledPin`, dando a essa variável um valor 10.

Em seguida, passamos à função `setup()` que simplesmente define o modo para o pino digital 10 como saída (OUTPUT).

No laço principal do programa que estabelecemos pino digital 10 HIGH, envio de 5 V. Então vamos esperar por um segundo e, em seguida, desligar o 5 V do pino 10, antes de esperar outro 1 s. O laço, em seguida, começa do início e o LED, portanto, liga e desliga continuamente enquanto o Arduino estiver energizado.

Agora que você sabe isso, você pode modificar o código para ligar o LED em um período diferente de tempo e também desligá-lo por um período de tempo diferente.

Por exemplo, se quisermos que o LED permaneça aceso por dois segundos, em seguida, desligado meio segundo nós poderíamos fazer o seguinte:

```
void loop() {
  digitalWrite(ledPin, HIGH);
  delay(2000);
  digitalWrite(ledPin, LOW);
  delay(500);
}
```

ou talvez você gostaria que o LED fique desligado por 5 segundos e, em seguida, piscar rapidamente (250 ms), como o LED indicador de um alarme de carro, então você poderia fazer isso:

```
void loop() {
  digitalWrite(ledPin, HIGH);
  delay(250);
  digitalWrite(ledPin, LOW);
  delay(5000);
}
```

ou fazer o LED ligar e desligar muito rápido:

```
void loop() {
  digitalWrite(ledPin, HIGH);
  delay(50);
  digitalWrite(ledPin, LOW);
  delay(50);
}
```

Variando entre ligar e desligar o LED, você pode criar qualquer efeito que desejar.

Projeto 2 - Código Morse SOS sinalizador

Para este projeto, vamos deixar o circuito exatamente do mesmo jeito configurado como no Projeto 1, mas teremos algumas diferenças no código para fazer o display LED emitir uma mensagem em código Morse. Neste caso, faremos com que o LED sinalize as letras S.O.S., que é o código Morse internacional de sinal de socorro.

O código Morse é um tipo de codificação que transmite letras e números usando padrões ON e OFF. Ele é, portanto, bem adequado para os nossos sistemas digitais como podemos transformar um LED ligado e fora do padrão necessário para soletrar uma palavra ou uma série de caracteres.

Neste caso estaremos sinalizando S.O.S. que no alfabeto do código Morse é de três pontos (flash curto), seguido por três traços (flash de comprimento longo), seguido por três pontos novamente.

Podemos, portanto, agora com código do nosso esboço piscar o LED ON e fora deste padrão, sinalizando SOS. Digite o código para criar uma nova configuração. Verifique se o seu código é livre de erros e, em seguida, envie-o para o seu Arduino.

Se tudo correr bem você vai ver agora o LED em flash o código sinal Morse SOS, aguarde 5 segundos, em seguida, repita. Então, vamos dar uma olhada neste código e descobrir como ele funciona.

```
// Projeto 2 - Codigo Morse SOS
// LED connected to digital pin 10
int ledPin = 10;
// run once, when the sketch starts
void setup()
{
  // sets the digital pin as output
  pinMode(ledPin, OUTPUT);
}
// run over and over again
void loop()
{
  // 3 dits
  for (int x=0; x<3; x++) {
    digitalWrite(ledPin, HIGH); // sets the
LED on
    delay(150); // waits for 150ms
    digitalWrite(ledPin, LOW); // sets the
LED off
    delay(100); // waits for 100ms
  }
  // 100ms delay to cause slight gap
between letters
  delay(100);
  // 3 traços
  for (int x=0; x<3; x++) {
    digitalWrite(ledPin, HIGH); // sets the
LED on
    delay(400); // waits for 400ms
    digitalWrite(ledPin, LOW); // sets the
LED off
    delay(100); // waits for 100ms
  }
  // 100ms delay to cause slight gap
between letters
  delay(100);
  // 3 dits again
  for (int x=0; x<3; x++) {
    digitalWrite(ledPin, HIGH); // sets the
LED on
```

```

    delay(150); // waits for 150ms
    digitalWrite(ledPin, LOW); // sets the
LED off
    delay(100); // waits for 100ms
}
// wait 5 seconds before repeating the
SOS signal
delay(5000);
}

```

Projeto 2 – Revisão do código

Assim, a primeira parte do código é idêntica ao último projeto em que inicializa um pino variável e define-se o pino 10 para ser uma saída. No código principal, podemos ver o mesmo tipo de declaração para transformar os LEDs ligados e desligados por um período definido de tempo, mas desta vez as declarações estão dentro de três blocos de código separados.

O primeiro bloco é o que gera três piscadas

```

for (int x=0; x<3; x++) {
digitalWrite(ledPin, HIGH);
delay(150);
digitalWrite(ledPin, LOW);
delay(100);
}

```

Podemos ver que o LED é ligado por 150 ms e então desligado 100 ms. Podemos ver que aquelas declarações que estão dentro de um conjunto de chaves são, portanto, um bloco de código separado. Mas, quando executamos o *sketch*, podemos ver que a luz pisca três vezes e não apenas uma vez. Isso é feito usando o laço **for**:

```

for (int x=0; x<3; x++) {

```

Esta declaração é o que torna o código dentro de seu código executar três vezes. Existem 3 parâmetros para dar o laço. Estes são de inicialização, incremento e condição. A inicialização acontece pela primeira vez e exatamente uma vez. Cada vez através do loop, a condição é testada e, se ele é verdadeiro, o bloco de declaração, e o incremento são executados, em seguida, a condição é testada novamente. Quando a condição se torna falsa, o laço termina. Então, primeiro precisamos inicializar uma variável para ser o início do loop. Neste caso, configurar a variável X e configurá-lo para zero.

```

int x=0;

```

Em seguida, definir uma condição para decidir quantas vezes o código no loop será executado.

```

x < 3;

```

Neste caso, o laço será executado enquanto x for menor do que (<) 3. O código dentro de um laço for será sempre executado uma vez e não importa como a condição é definida. O símbolo < é conhecido como um operador de comparação. Eles são usados para tomar decisões dentro de seu código e comparar dois valores. Os símbolos utilizados são:

```

== (equal to)
!= (not equal to)
< (less than)
> (greater than)
<= (less than or equal to)
>= (greater than or equal to)

```

No nosso código estamos comparando x com o valor 3 para ver se é menor do que ele. Se x for menor que 3, então o código do bloco será repetido novamente. A declaração final é x++: esta é uma instrução para aumentar o valor de x em 1. Poderíamos ter digitado também x = x+1, o que faria atribuir a x o valor de x + 1. Observe que não há necessidade de colocar um ponto e vírgula após esta declaração final, em loop. Você pode fazer operações matemáticas simples, usando os símbolos +, -, * e / (adição, subtração, multiplicação e divisão). Por exemplo:

```

1 + 1 = 2
3 - 2 = 1
2 * 4 = 8
8 / 2 = 4

```







Assim, o nosso laço para inicializar compara o valor de x com 3, em seguida, é executado o código dentro do bloco. Então aumenta o incremento, neste caso adiciona 1 a x.

Em seguida, verifica se a condição foi atendida, o que é se x é menor do que 3, e se repete. Então, agora podemos ver em nosso código que há três laços, que circulam três vezes e exibe os pontos, no próximo repete três vezes e exibe os traços, então segue a repetição dos pontos novamente. Deve-se notar que a variável x tem um alcance local, o que significa que só pode ser visto pelo código dentro seu bloco. A menos que você inicialize-o antes da função `setup()`, caso em que tem um alcance global e pode ser visto por todo o programa. Se você tentar acessar x fora do laço você obterá um erro. Entre cada ciclo “for” existe um pequeno atraso para fazer uma pequena pausa visível entre as letras de SOS. Finalmente, o código espera 5 segundos antes do principal laço do programa e recomeça desde o início.

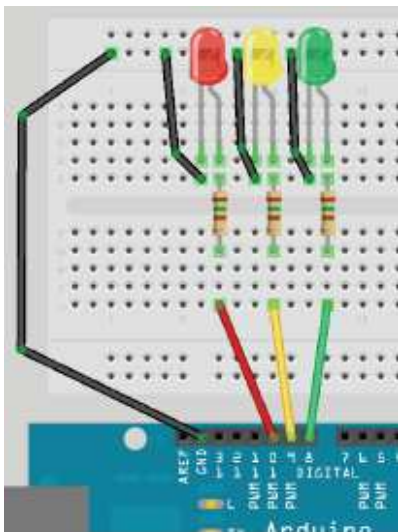
Projeto 3 – Semáforo

Agora vamos criar um conjunto de semáforos que vai mudar de verde para amarelo e depois vermelho e depois volta novamente, depois de um conjunto período de tempo utilizando os quatro estados do sistema. Este projeto poderia ser utilizado em uma estrada de ferro modelo ou para uma minicidade de brinquedo.

Você precisará de:

Breadboard	
Red Diffused LED	
Yellow Diffused LED	
Green Diffused LED	
3 x 220Ω Resistors	
Jumper Wires	

Montagem:



Desta vez vamos ligar três LEDs com o anodo de cada um indo para pinos digitais 8, 9 e 10, através de três resistores de 150Ω. Tomamos um fio de terra para o trilho solo no topo da placa de montagem e um longo fio que vai a partir da perna do catodo de cada LED para a trilha comum. Digite o código a seguir, verificar e fazer o upload. Se você leu os Projetos 1 e 2, então este código deverá ser auto-explicativo assim como o hardware.

```
// Projeto 3 - Semáforo
int ledDelay = 10000; // delay in between
changes
int redPin = 10;
int yellowPin = 9;
int greenPin = 8;
void setup() {
  pinMode(redPin, OUTPUT);
  pinMode(yellowPin, OUTPUT);
  pinMode(greenPin, OUTPUT);
}
void loop() {
  // turn the red light on
  digitalWrite(redPin, HIGH);
  delay(ledDelay); // wait 5 seconds
  digitalWrite(yellowPin, HIGH); // turn on
  yellow
  delay(2000); // wait 2 seconds
  digitalWrite(greenPin, HIGH); // turn
  green on
  digitalWrite(redPin, LOW); // turn red
  off
  digitalWrite(yellowPin, LOW); // turn
  yellow off
  delay(ledDelay); // wait ledDelay
  milliseconds
  digitalWrite(yellowPin, HIGH); // turn
  yellow on
  digitalWrite(greenPin, LOW); // turn
  green off
  delay(2000); // wait 2 seconds
  digitalWrite(yellowPin, LOW); // turn
  yellow off
  // now our loop repeats
}
```






No próximo projeto, vamos adicionar a este projeto um conjunto de luzes de pedestres e também adicionar um botão de pulso para fazer as luzes interagirem.

Projeto 4 - Semáforo interativo

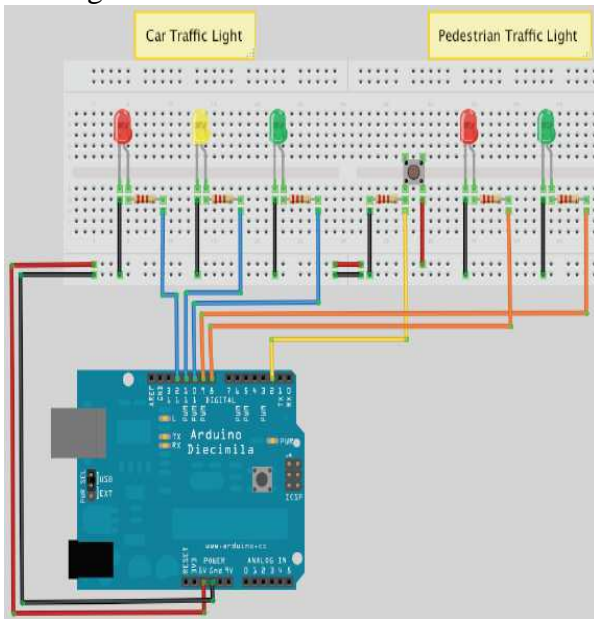
Desta vez, vamos estender o projeto anterior de modo a incluir um conjunto de luzes de pedestre. Pressionar o botão para atravessar a rua e o Arduino reagirá quando o botão está pressionado, alterando o estado das luzes para fazer os carros pararem e permitir que o pedestre atravessasse com segurança.

Pela primeira vez, somos capazes de interagir com o Arduino e fazer com que ele faça alguma coisa quando nós alterarmos o seu estado (pressione para alterar o estado de aberto para fechado). Neste projeto você também vai aprender como criar suas próprias funções.

O que você precisará:

2 x Red Diffused LED's	
Yellow Diffused LED	
2 x Green Diffused LED's	
6 x 150Ω Resistors	
Tactile Switch	

Montagem



Digite o código da página seguinte, verificar e carregá-lo.

Quando executar o programa você verá que o semáforo para carros começa no verde para permitir que os carros passem e a luz de pedestres está no vermelho. Quando pressionar o botão, o programa verifica se pelo menos 5 segundos se passaram desde a última vez que as luzes foram alteradas (para permitir o tráfego entrar em movimento), e assim passa a execução do código para a função que nós criamos chamada *changeLights()*. Nesta função, as luzes do semáforo vão de verde para amarelo e depois vermelho, em seguida as luzes de pedestres tornam-se verde. Após um período de tempo fixado no Crosstime variável (tempo suficiente para permitir que os pedestres atravessassem) a luz verde para pedestres irá acender e apagar como um aviso para os pedestres para atravessar de pressa e que as luzes estão prestes a mudar de volta ao vermelho. Então, o semáforo de pedestres retorna para o vermelho e as luzes do semáforo para veículos vão do vermelho para o verde e o

tráfego pode continuar. O código deste projeto é semelhante ao do projeto anterior. No entanto, existem algumas declarações novas e conceitos que foram introduzidos. Então, vamos ver:

```
// Projeto 4 - Semáforo interativo
// Atribue as luzes para os carros
int carRed = 12;
int carYellow = 11;
int carGreen = 10;
int pedRed = 9;
// Atribue as luzes do pedestre
int pedGreen = 8;
int button = 2; // Pino do botao
int crossTime = 5000; // Tempo travessia
unsigned long changeTime; // Tempo desde
que o botao foi pressionado
```

```
void setup() {
pinMode(carRed, OUTPUT);
pinMode(carYellow, OUTPUT);
pinMode(carGreen, OUTPUT);
pinMode(pedRed, OUTPUT);
pinMode(pedGreen, OUTPUT);
pinMode(button, INPUT);
// Acende a luz verde para os carros
digitalWrite(carGreen, HIGH);
digitalWrite(pedRed, HIGH);
}
```

```
void loop() {
int state = digitalRead(button);
/* Verifica se o botao foi pressionado ha
mais de 5 s desde a ultima vez */
if (state == HIGH && (millis() -
changeTime) > 5000) {
// Chama a funcao para alterar as luzes
changeLights();
}
}
```

```
void changeLights() {
digitalWrite(carGreen, LOW); // Desliga o
verde
digitalWrite(carYellow, HIGH); // Liga o
amarelo
delay(2000); // Aguarda 2 segundos
digitalWrite(carYellow, LOW); // Desliga
o amarelo
digitalWrite(carRed, HIGH); // Liga o
vermelho
delay(1000); // Aguarda 1 s de segurancia
digitalWrite(pedRed, LOW); // Desliga o
vermelho do pedestre
digitalWrite(pedGreen, HIGH); // Liga o
verde do pedestre
delay(crossTime); // Aguarda o tempo
prefixado para travessia do pedestre
// Pisca o verde do pedestre
for (int x=0; x<10; x++) {
digitalWrite(pedGreen, HIGH);
delay(250);
digitalWrite(pedGreen, LOW);
delay(250);
}
// Liga o vermelho do pedestre
digitalWrite(pedRed, HIGH);
delay(500);
}
```



```
digitalWrite(carYellow, HIGH); // Liga o
amarelo
digitalWrite(carRed, LOW); // Desliga o
vermelho
delay(1000);
digitalWrite(carGreen, HIGH);
digitalWrite(carYellow, LOW); // Desliga
o amarelo
// Registra o tempo desde a ultima
mudanca do semaforo
changeTime = millis();
// Retorna ao laco do programa principal
}
```

Projeto 4 – Descrição do código

Você vai entender e reconhecer a maioria do código neste projeto a partir dos projetos anteriores. No entanto, vamos dar uma olhada em algumas palavras-chave e conceitos novos que foram introduzidas neste *sketch*.

```
unsigned long changeTime;
```

Aqui temos um novo tipo de dado (`unsigned long`) para uma variável. Anteriormente criamos tipos de dados inteiros, que podem armazenar um número entre -32.768 e +32.767. Agora, criamos um tipo armazenador de dados longo, que pode armazenar um número de -2.147.483.648 a +2.147.483.647. No entanto, especificou-se um `unsigned long`, que significa que a variável não pode armazenar números negativos, de tal maneira que nos dá um intervalo de 0 a 4.294.967.295. Se fosse usar um número inteiro para armazenar o período de tempo desde a última mudança de luzes, só iria ter um tempo máximo de 32 segundos antes de a variável inteira atingir um número maior do que ele pode armazenar. Como numa faixa de pedestres é improvável que seja utilizado tempo pequeno, de no máximo 32 s, não queremos o programa falhando devido a nossa variável transbordando quando se tenta armazenar um número muito maior que o tipo de variável inteira. É por isso que usamos um tipo de variável `unsigned long` para obter um comprimento de tempo longo o suficiente para travessia de pedestres.

$$4.294.967.295 \times 1 \text{ ms} = 4.294.967 \text{ segundos}$$

$$4.294.967 \text{ segundos} = 71.582 \text{ minutos}$$

$$71.582 \text{ minutos} = 1.193 \text{ horas}$$

$$1.193 \text{ horas} = 49 \text{ dias}$$

Como é bastante difícil que em uma faixa de pedestres o botão não seja pressionado pelo menos uma vez em 49 dias, não devemos ter um problema como este de dados. Você pode perguntar por que nós não temos só um conjunto de dados que pode armazenar grandes números de

tempo e tudo ser feito com ele. Bem, a razão é porque as variáveis ocupam espaço na memória assim como os números e isso pode prejudicar o armazenamento de variáveis.

Em seu PC ou laptop você não terá que se preocupar com isso, mas em um microcontrolador pequeno como o Atmega328, é essencial usarmos apenas os menores dados variáveis necessários para o nosso propósito.

Existem vários tipos de dados que podemos usar em nossos *sketches* e estes são:

Tipo de dado	RAM	Faixa
void keyword	N.D.	N.D.
boolean	1 byte	0 e 1 (falso/verdade)
byte	1 byte	0 a +255
char	1 byte	-128 a +127
unsigned char	1 byte	0 a +255
int	2 byte	-32768 a +32767
unsigned int	2 byte	0 a +65535
word	2 byte	0 a +65535
long	4 byte	-2147483548 a +2147483547
unsigned long	4 byte	0 a +4294967295
float	4 byte	-3,4028235.10 ⁺³⁸ a +3,4028235.10 ⁺³⁸
double	4 byte	-3,4028235.10 ⁺³⁸ a +3,4028235.10 ⁺³⁸
string	1 byte + x	Matriz de caracteres
array	1 byte + x	Coleção de variáveis

Em cada tipo de dado usa-se certa quantidade de memória do Arduino, como pode ser observado na Tabela acima. Algumas variáveis usam apenas 1 byte de memória e outras usam 4 bytes ou mais. O microcontrolador ATmega168 tem 1 kB (1000 bytes) e o Atmega328 tem 2 kB (2000 bytes) de SRAM; isto não é muito e em programas de grande porte com muitas variáveis você poderia facilmente usar muita memória. A partir da Tabela acima, podemos ver claramente que o uso do tipo de dados `int` é um desperdício, pois utiliza 2 bytes e pode armazenar um número inteiro até 32.767. Como temos usado `int` para armazenar o número do nosso pino digital, que só vai até 13 no Arduino Uno (e até 54 no Arduino Mega), temos usado mais memória do que o necessário. Poderíamos ter economizado memória utilizando o tipo de dados `byte`, que pode armazenar um número entre 0 e 255, que é mais do que suficiente para armazenar o número de pinos de entrada/saída (E/S) do Arduino.

Em seguida, temos:

```
pinMode(button, INPUT);
```

Isto diz ao Arduino que queremos usar o pino digital 2 (`button = 2`) como `INPUT`. Nós vamos usar o pino 2 para o botão pressionado e o seu modo deve ser definido como entrada. No laço

principal do programa verifica o estado do pino digital 2 com esta declaração:

```
int state = digitalRead(button);
```

Este inicializa um inteiro (sim, um desperdício e nós deveríamos usar uma variável booleana), em seguida, define o valor de estado para ser o valor do pino digital 2. A declaração `digitalRead` lê o estado do pino digital dentro do parêntese e retorna para os números inteiros que foram atribuídos a ele.

Podemos, então, verificar o valor no estado para ver se o botão foi pressionado ou não.

```
if (state == HIGH && (millis() -
changeTime) > 5000) {
  changeLights();
}
```

A instrução `if` é um exemplo de uma estrutura de controle e sua finalidade é verificar se uma determinada condição foi cumprida ou não, e se sim, para executar o código dentro do seu bloco de código. Por exemplo, se quiséssemos transformar um LED em uma variável chamada `x` e subir acima do valor de 500 podemos escrever:

```
if (x>500) {digitalWrite(ledPin, HIGH);
```

Quando lemos um pino digital usando o comando `digitalRead`, o estado do pino irá ser `HIGH` ou `LOW`. Assim, o comando `if` em nosso *sketch* é:

```
if (state == HIGH && (millis()-
changeTime) > 5000)
```

O que estamos fazendo aqui é verificar se duas condições foram satisfeitas. A primeira é que a variável chamada estado é elevada. Se o botão foi pressionado vai ser elevada como já definido para ser o valor lido no pino digital 2. Estamos verificando também se o valor de `millis()-changeTime` é maior que 5000 (utilizando os operadores lógicos `&&`). A função `millis()` é uma construção dentro da linguagem Arduino e retorna ao número de milissegundos desde que o Arduino começou a executar o programa atual.

Subtraindo o valor da variável `changeTime` do valor `millis()` atual, podemos verificar se 5 segundos se passaram desde a última `changeTime()` definida. O cálculo do `millis()-changeTime` é colocado dentro do seu próprio conjunto de parênteses para assegurar a comparação do valor de estado e o resultado deste cálculo e não o valor de milissegundos por conta própria.

O estado:

```
state == HIGH
```

e o cálculo é um exemplo de um operador booleano. Para ver o que nós queremos dizer com isso, vamos dar uma olhada em todos os operadores booleanos.

<code>&&</code>	E lógico (AND)
<code> </code>	OU lógico (OR)
<code>!</code>	NÃO (NOT)

Estas são afirmações lógicas que podem ser usadas para testar diversas condições em `if`. O operador `&&` significa verdadeiro se ambos os operadores são verdadeiros, por exemplo,

```
if (x == 5 && y == 10) {...
```

Esta declaração `if` vai executar o seu código `se` e somente se `x` for igual a 5 e também se `y` for igual a 10. `||` significa verdadeiro se o operando for verdadeiro. Por exemplo

```
if (x == 5 || y == 10) {...
```

Será executado se `x` for 5 **ou** se `y` for 10.

```
if (!x) {...
```

Será executado se `x` for falso, ou seja igual a zero. Você pode também reunir condições com parênteses, por exemplo:

```
if (x==5 && (y == 10 || z == 25)) {...
```

Neste caso, as condições dentro do parêntese interno são processadas separadamente e tratadas como uma única condição e, em seguida, comparada à segunda condição. Assim, se quisermos desenhar uma tabela verdade simples desta afirmação, podemos ver como funciona.

x	y	z	Falso/Verdade?
4	9	25	FALSO
5	10	24	VERDADE
7	10	25	FALSO
5	10	25	VERDADE

O comando dentro da declaração

```
if (state == HIGH && (millis() -
changeTime) > 5000) {
```

é

```
changeLights();
```

Este é um exemplo de uma chamada de função. Uma função é simplesmente um bloco de códigos separado que foi dado um nome. No entanto, nas funções podem ser passados os parâmetros e / ou dados de retorno também.

Vamos entrar em mais detalhes mais tarde, sobre a passagem de parâmetros e retorno de dados de funções.



Quando `changeLights()` é chamado, o código executa saltos a partir da linha atual para a função, executa o código dentro dessa função e, em seguida, retorna ao ponto no código depois que a função foi chamada.

Assim, neste caso, se as condições previstas na instrução `if` forem satisfeitas, em seguida o programa executa o código dentro da função e depois retorna para a próxima linha após `changeLights()`; em `if`. O código dentro da função simplesmente muda as luzes de veículos para vermelho, em seguida, torna a luz de pedestre verde. Após um período de tempo definido pela variável `Crosstime` a luz pisca um tempo pequeno para alertar o pedestre que seu tempo está prestes a esgotar-se; então a luz de pedestre vai para o vermelho e a luz de veículo vai de vermelho para verde, e assim retorna para o seu estado normal. O programa principal simplesmente verifica continuamente se o botão de pedestre foi pressionado ou não e se há tempo suficiente desde que as luzes de pedestres foram acionadas da última vez (maior do que 5 segundos).

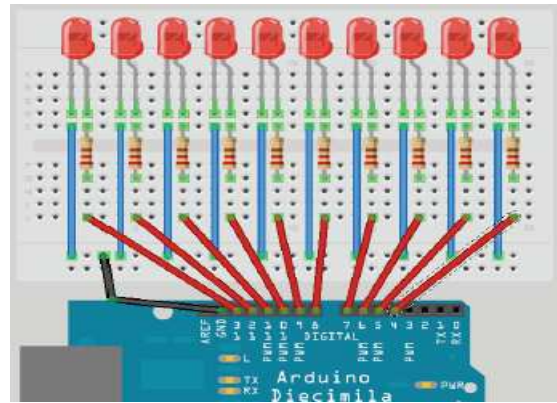
Projeto 5 - Efeito perseguição com LEDs

Faremos agora uma seqüência de LEDs (10 no total) fazer um efeito de perseguição de LED e dessa maneira introduzir o conceito muito importante de matrizes.

O que você precisa

10 x Red Diffused LED's	
10 x 220Ω Resistors	

Monte o circuito:



Insira o código:

```
// Projeto 5 - Efeito perseguição com LED
// Criar matriz para os pinos dos LEDs
byte ledPin[] = {4, 5, 6, 7, 8, 9, 10,
11, 12, 13};
int ledDelay(65); // Atraso entre mudança
int direction = 1;
int currentLED = 0;
unsigned long changeTime;

void setup() {
// Configurar todos os pinos de saída
for (int x=0; x<10; x++) {
pinMode(ledPin[x], OUTPUT); }
changeTime = millis();
}

void loop() {
// if it has been ledDelay ms since last
change
if ((millis() - changeTime) > ledDelay) {
changeLED();
changeTime = millis();
}
}

void changeLED() {
// turn off all LED's
for (int x=0; x<10; x++) {
digitalWrite(ledPin[x], LOW);
}
// turn on the current LED
digitalWrite(ledPin[currentLED], HIGH);
// increment by the direction value
currentLED += direction;
// change direction if we reach the end
if (currentLED == 9) {direction = -1;}
if (currentLED == 0) {direction = 1;}
}
```

Projeto 5 – Descrição do código

Nossa primeira linha no *sketch* é

```
byte ledPin[] = {4, 5, 6, 7, 8, 9,
10, 11, 12, 13};
```

Esta é uma declaração de uma variável do tipo matriz de dados. Uma matriz é uma coleção de variáveis que são acessados usando um número de índice. Em nosso esboço nós declaramos um array de byte de tipo de dados e chamamos de `ledPin`. Temos então iniciada a matriz com 10 valores, que são de 4 pinos digitais até o número 13. Para acessar um elemento da matriz basta

simplesmente se referir ao número de índice do elemento. No entanto, em nossa matriz de elemento 10, o índice dos números são de 0 a 9. Neste caso, o 3º elemento (`ledPin[2]`) tem o valor de 6 e o 7º elemento (`ledPin[6]`) tem um valor de 10. Você tem que dizer o tamanho da matriz, se não inicializar com os primeiros dados. No nosso esboço não fizemos explicitamente a escolha de um tamanho, como o compilador é capaz de contar os valores que já atribuídos para a matriz e trabalhar com tamanho de 10 elementos. Se tivéssemos declarado a matriz, mas não inicializado com valores ao mesmo tempo, seria preciso declarar uma dimensão, para exemplo, poderíamos ter feito isso:

```
byte ledPin[10];
```

Em seguida carregado dados nos elementos mais tarde. Para recuperar um valor do array faríamos algo como este:

```
x = ledPin[5];
```

Neste exemplo `x` teria agora um valor 8. Em nosso programa, começamos declarando e inicializando um array que armazenou 10 valores, que são os pinos digitais utilizados para as saídas dos 10 LEDs. Em nosso laço verificamos que alguns milissegundos do `ledDelay` se passam desde a última mudança de LEDs e, assim acontece o controle para a nossa função. A função que criamos é

```
void changeLED() {
for (int x=0; x<10; x++) {
    digitalWrite(ledPin[x], LOW);
}
digitalWrite(ledPin[currentLED], HIGH);
currentLED += direction;
if (currentLED == 9) {direction = -1;}
if (currentLED == 0) {direction = 1;}
}
```

E o trabalho desta função é passar todos os LEDs para desligados e em seguida, liga-los (isso é feito tão rápido que você não verá que isso acontece), através da variável `currentLED`.

Esta variável, em seguida, tem a direção adicionada. Como a direção só pode ser +1 ou -1, então o número irá aumentar (+1) ou diminuir (-1) (`currentLED + (-1)`).

Temos então, um comando `if` para verificar se atingimos o fim da linha de LEDs e, em seguida, inverter a direção.

Ao alterar o valor de `ledDelay` você pode fazer com que o LED vá de frente para trás em velocidades diferentes. Tente valores diferentes para ver o que acontece.


No entanto, você tem que parar o programa e manualmente alterar o valor de `ledDelay` e então refazer o upload do código alterado para ver as alterações. Não seria mais agradável ser capaz de ajustar a velocidade, enquanto o programa está funcionando?

Sim, seria. Então vamos fazer exatamente isso no próximo projeto, através da introdução de uma forma que interaja com o programa e ajuste a velocidade usando um potenciômetro.

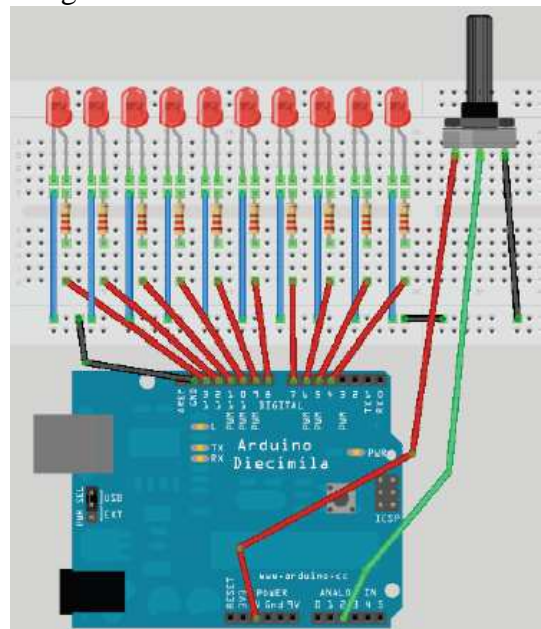
Projeto 6 – Efeito interativo de perseguição com LEDs

Iremos agora fazer uma sequência de LEDs (10 no total) produzindo um efeito de perseguição de LED, que também será maneira introduzir o conceito de matrizes.

O que você precisa

Parts from previous project plus....	
4K7 Potentiometer	

Montagem:



Este é o mesmo circuito do Projeto 5, mas com a adição de um potenciômetro, que estará conectado à fonte de 5 V e à porta analógica do Pino 2.

Projeto 6 - Visão geral do código

```
// Cria a matriz de pinos dos LEDs
```

```

byte ledPin[] = {4, 5, 6, 7, 8, 9, 10,
11, 12, 13};
int ledDelay; // Atraso entre mudancas
int direction = 1;
int currentLED = 0;
unsigned long changeTime;
int potPin = 2; // Pino de entrada do
potenciometro

void setup() {
// Configura os pinos de saida
for (int x=0; x<10; x++) {
pinMode(ledPin[x], OUTPUT); }
changeTime = millis();
}

void loop() {
// Leitura do valor do pot
ledDelay = analogRead(potPin);
// Verifica se ocorreu ledDelay desde a
ultima mudanca
if ((millis() - changeTime) > ledDelay) {
changeLED();
changeTime = millis();
}
}

void changeLED() {
// Desliga todos os LEDs
for (int x=0; x<10; x++) {
digitalWrite(ledPin[x], LOW);
}
// Desliga o LED corrente
digitalWrite(ledPin[currentLED], HIGH);
// Mude a direcao
currentLED += direction;
// Mude a direcao se chegar ao final
if (currentLED == 9) {direction = -1;}
if (currentLED == 0) {direction = 1;}
}

```

O código para este projeto é quase idêntico ao do projeto anterior. Nós simplesmente acrescentamos um potenciômetro ao nosso hardware e o código tem adições para habilitar-nos a ler os valores do potenciômetro e utilizá-los para ajustar a velocidade do LED com o efeito de perseguição. Primeiro, devemos declarar uma variável para o pino do potenciômetro:

```
int potPin = 2;
```

O potenciômetro estará conectado ao pino analógico 2. Para ler o valor de um pino analógico usamos o comando `analogRead`.

O Arduino tem seis portas analógicas de entrada/saída de 10-bits cada, para o conversor digital (vamos discutir os bits mais tarde). Isto significa que os pinos analógicos podem ler tensões entre 0 e 5 volts em valores inteiros entre 0 (0 volts) e 1023 (5 volts). Isso nos dá uma resolução de 5 volts/1024 unidades ou 0,0049 volts (4,9 mV) por unidade.

Precisamos definir o nosso atraso usando o potenciômetro; assim, vamos simplesmente usar os valores de leitura direta a partir do pino para ajustar o atraso entre 0 e 1023 milissegundos. Fazemos isso para ler diretamente o valor do pino

do potenciômetro em `ledDelay`. Observe que nós não precisamos definir um pino analógico para ser uma entrada ou saída como nós precisamos de um pino digital.

```
ledDelay = analogRead(potPin);
```

Essa leitura é feita durante o funcionamento do nosso circuito principal e por isso é constantemente lido e ajustado. Ao transformar o botão você pode ajustar o valor de atraso entre 0 e 1023 milissegundos (ou pouco mais de um segundo) e portanto, tem total controle sobre a velocidade do efeito. OK, vamos descobrir o que é um potenciômetro e como funciona.

Projeto 6 - Visão geral do hardware

A única peça adicional de hardware utilizado neste projeto foi o potenciômetro 4K7 (4700Ω).



O potenciômetro é simplesmente um resistor ajustável com um intervalo de 0 a um valor definido (escrito no lado de baixo dele). No kit utilizou-se um potenciômetro de 4K7 ohm (ou 4.700 Ω), que significa que a sua faixa de resistência varia de 0 a 4.700 ohm. O potenciômetro tem três pernas. Ao conectar duas pernas adjacentes no potenciômetro, ele torna-se um resistor variável. Ao ligar todas as três pernas e aplicando uma voltagem nas pernas mais externas, o potenciômetro torna-se um divisor de tensão. Isto é o que nós utilizamos em nosso circuito. Um lado é ligado a terra, a outra a 5 V e do centro fixa o nosso pino analógico. Ao ajustar o botão, uma tensão entre 0 e 5 V irá ser distribuída a partir do pino central e podemos ler o valor que a tensão do Pino 2 analógico distribuiu e usar o seu valor para alterar a taxa de atraso do luz efeito.

O potenciômetro pode ser muito útil para proporcionar um meio para ajustar um valor de 0 a um valor definido de resistência. Por exemplo, o volume de um rádio ou o brilho de uma lâmpada. Na verdade, interruptores *dimmer* (que variam a intensidade da luz) para as lâmpadas de sua casa são um tipo de potenciômetro.

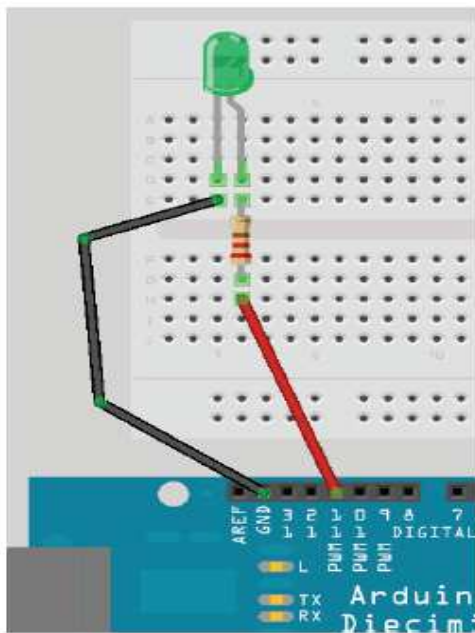
Projeto 7 - Lâmpada pulsante

Vamos agora aprofundar ainda mais o método de controle dos LEDs. Até agora, vimos simplesmente o LED ligado ou desligado. Que tal ser capaz de ajustar o brilho, também? Podemos fazer isso com um Arduino? Sim, podemos. É hora de voltar ao básico.

O que você precisa:

Green Diffused LED	
220Ω Resistor	

Conecte:



Digite o código no IDE

```
// Projeto 7 - Lampada pulsante
int ledPin = 11;
float sinVal;
int ledVal;
void setup() {
  pinMode(ledPin, OUTPUT);
}
void loop() {
  for (int x=0; x<180; x++) {
    // Converte graus para radianos
    // entao calcula os valores de seno
    sinVal = (sin(x*(3.1412/180)));
    ledVal = int(sinVal*255);
    analogWrite(ledPin, ledVal);
    delay(25);
  }
}
```

Verifique e faça o upload. Você vai ver agora o LED pulsar e desligar de forma contínua. Em vez de um simples liga/desliga (*on/off*) de estado, o programa está agora ajustando o brilho do LED. Vamos descobrir como isso funciona.

Projeto 7 – Descrição do código

O código para este projeto é muito simples, mas requer alguma explicação.

Primeiro configuramos as variáveis para `ledPin` como `float` (tipo de dados ponto flutuante) para um valor de onda senoidal e `ledVal` que vai conter o valor inteiro e enviar ao Pino 11.

O conceito aqui é que estamos criando uma onda senoidal e fazendo o brilho do LED seguir o caminho dessa onda. Isto é o que faz pulsar a luz e dessa forma desaparecer o brilho total e reacender novamente.

Usamos a função `sin()`, que é uma função matemática para realizar o trabalho angular através do seno de um ângulo. Precisamos transformar a função de graus a radianos. Nós temos um laço `for-end` que vai de 0 a 179, pois não queremos ir além dos valores positivos. Acima deste valor nos levaria a valores negativos e o valor do brilho necessita ser entre 0 e 255.

A função `sin()` requer o ângulo em radianos e não em graus, de modo que a equação $x*3.1412/180$ irá converter o ângulo em radianos. Em seguida, transferir o resultado para `ledVal`, multiplicando-o por 255 e nos dará o valor do brilho. O resultado da função `sin()` será um número entre -1 e 1; por isso precisamos multiplicar por 255 para nos dar o valor do brilho. Nós convertemos o valor em ponto flutuante de `Sinval` em um número inteiro através da função `int()` na instrução.

```
ledVal = int(sinVal*255);
```

Em seguida, enviar esse valor para o Pino 11 utilizando a afirmação:

```
analogWrite(ledPin, ledVal);
```

Mas, como podemos enviar um valor analógico para um pino digital? Bem, se dermos uma olhada no Arduino e olharmos para os pinos digitais você poderá ver que seis desses pinos (pino 3, 5, 6, 9, 10 e 11) têm PWM escrito ao lado deles. Esses pinos diferem dos pinos digitais restantes porque são capazes de enviar um sinal PWM.

PWM é a sigla em inglês de **modulação por largura de pulso** (*pulse width modulation*). É uma técnica para obtenção de resultados analógicos a partir de sinais digitais. Esses pinos do Arduino enviam uma onda quadrada trocando o sinal do pino dentro muito rápido (*on/off*). O padrão *on/off* pode simular uma tensão variando entre 0 e 5 V. Ele faz isso alterando a quantidade

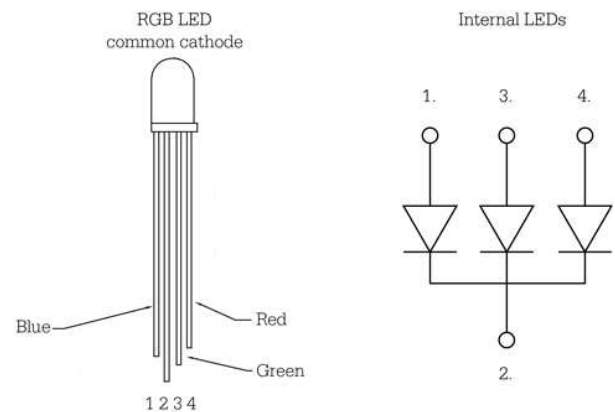
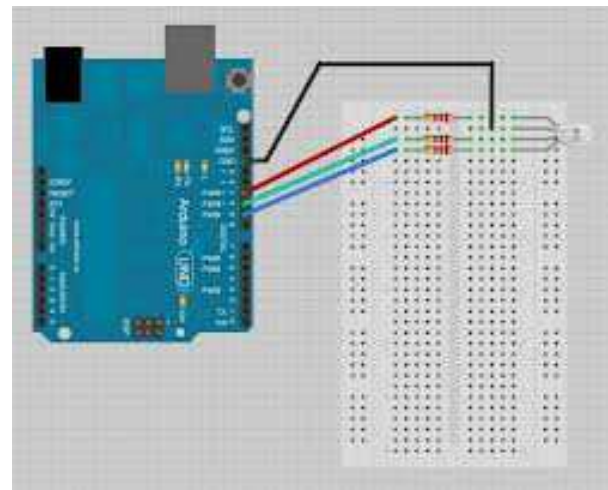
de tempo que a saída permanece elevada (*on*) versus baixa (*off*). A duração do tempo é conhecida como largura de pulso.

Por exemplo, se for enviado o valor 0 para o pino 11 usando `analogWrite()`, o período ON seria zero, ou teria um ciclo de 0%. Se fosse enviado um valor de 64 (correspondendo a 25% de 255) o pino seria ON para 25% do tempo e OFF para o restante 75% do tempo. O valor 191 corresponderia a um ciclo de trabalho de 75%, enquanto que um valor de 255 corresponderia a um ciclo de trabalho de 100%. Os impulsos ocorrem a uma frequência de aproximadamente 500 Hz ou a duração de 2 ms cada.

Assim, podemos ver em nosso *sketch* que o LED está sendo ligado e desligado muito rápido. Se o ciclo de trabalho for de 50% (valor de 127), o LED irá pulsar de forma ligado e desligado em 500 Hz e mostrará a metade do brilho máximo. Basicamente é uma ilusão de ótica que podemos usar a nosso favor, permitindo utilizar pinos digitais para a saída de um valor simulando um sinal analógico para o nosso LED.

Note que apesar de apenas seis dos pinos terem a função PWM, você pode facilmente escrever um programa para fornecer uma saída PWM a todos os pinos digitais, se desejar.

Mais tarde, vamos revisitar a função PWM e utilizá-la para criar tons audíveis usando uma sirene piezelétrica.



Conecte o LED RGB, que poderá formar diversas cores, da forma que desejar apenas alternando a intensidade luminosa de cada cor.

Projeto 8 – Lâmpada do humor

No projeto anterior, vimos que podemos ajustar o brilho de um LED utilizando as capacidades de PWM do chip Atmega.

Vamos agora aproveitar essa capacidade utilizando um LED vermelho, verde e azul (RGB) e pela mistura das cores criarmos qualquer cor que desejarmos.

A partir disso, vamos criar uma lâmpada de humor semelhante ao tipo que você vê à venda em todo o lugar hoje em dia.

O que você precisa

LED tricolor RGB	
3x220 Ω resistores	

Montagem:

```
// Projeto 8 - Lampada do humor
float RGB1[3];
float RGB2[3];
float INC[3];
int red, green, blue;
int RedPin = 11;
int GreenPin = 10;
int BluePin = 9;
void setup()
{
  Serial.begin(9600);
  randomSeed(analogRead(0));
  RGB1[0] = 0;
  RGB1[1] = 0;
  RGB1[2] = 0;
  RGB2[0] = random(256);
  RGB2[1] = random(256);
  RGB2[2] = random(256);
}
void loop()
{
  randomSeed(analogRead(0));
  for (int x=0; x<3; x++) {
    INC[x] = (RGB1[x] - RGB2[x])/256; }
  for (int x=0; x<256; x++) {
    red = int(RGB1[0]);
    green = int(RGB1[1]);
    blue = int(RGB1[2]);
    analogWrite (RedPin, red);
    analogWrite (GreenPin, green);
```

```

analogWrite (BluePin, blue);
delay(100);
RGB1[0] -= INC[0];
RGB1[1] -= INC[1];
RGB1[2] -= INC[2];
}
for (int x=0; x<3; x++) {
RGB2[x] = random(556)-300;
RGB2[x] = constrain(RGB2[x], 0, 255);
delay(1000);
}
}

```

Quando você executar o programa, verá as cores mudarem lentamente. Pronto, você fez a sua própria lâmpada do humor.

Descrição do código

Os LEDs que constituem a lâmpada humor são de cores vermelho, verde e azul. Da mesma forma que o seu monitor de computador é feito de cores vermelho minúsculo, verde e (RGB) pontos azuis, o mapa pode gerar diferentes cores, ajustando o brilho de cada um dos três LEDs de tal maneira que nos dê um valor RGB e tons de cores diferentes.

Um valor RGB de “255,0,0” nos daria o vermelho puro. O valor “0,255,0” daria verde puro e “0,0,255” a cor azul pura. Ao misturar essas cores dos LEDs, podemos obter diversas cores. Você ainda pode fazer diferentes cores, como nesta tabela com alguns exemplos de combinação de cores:

Red	Green	Blue	Colour
255	0	0	Red
0	255	0	Green
0	0	255	Blue
255	255	0	Yellow
0	255	255	Cyan
255	0	255	Magenta
255	255	255	White

Ao ajustar o brilho usando PWM, podemos obter também todas as outras cores. Ao colocar os LEDs juntos e misturando seus valores, o espectro de luz das três cores somadas faz uma única cor.

A gama total de cores que podemos fazer usando a saída PWM é de 0 a 255 é 16.777.216 cores (256x256x256), sendo este valor muito maior do que nós precisamos.

No código, começamos desligados (pontos com valor 0), declarando alguns pontos flutuantes (float) da matriz de pontos e também algumas

variáveis inteiras que irão armazenar os nossos valores RGB, bem como um valor de incremento.

```

float RGB1[3];
float RGB2[3];
float INC[3];
int red, green, blue;

```

Na função de configuração temos

```

randomSeed(analogRead(0));

```

O comando *randomSeed* é utilizado para criar números aleatórios (na verdade, pseudo aleatórios). Chips de computador não são capazes de produzir números aleatórios de modo que tendem a olhar para dados numa parte da sua memória que podem ser diferentes ou olhar para uma tabela de valores diferentes e utilizá-las como um número pseudo-aleatório. Neste caso, o do valor que será dado ao *randomSeed* é um valor lido a partir do pino analógico 0. Como nós não temos nada ligado ao pino analógico 0 tudo o que será lido é um número aleatório criado pelo ruído analógico.

Uma vez que tenhamos definido um início para o nosso número aleatório, esse poderá criar outro número usando a função *random()*. Em seguida, temos dois conjuntos de valores RGB armazenados em um elemento de três matrizes. RGB1 são os valores RGB que queremos que a lâmpada comece (neste caso, todos em zero ou desligados).

```

RGB1[0] = 0;
RGB1[1] = 0;
RGB1[2] = 0;

```

Em seguida, a matriz RGB2 é um conjunto de valores aleatórios RGB que queremos que a lâmpada tenha para produzir o efeito aleatório:

```

RGB2[0] = random(256);
RGB2[1] = random(256);
RGB2[2] = random(256);

```

Neste caso, teremos um conjunto de números aleatórios (256) que vai dar um número entre 0 e 255 inclusive (como o número sempre varia de zero para cima).

Se indicarmos um número para a função *random()*, ele então irá processar um valor entre 0 e 1 a menos do que o número indicado, por exemplo, *random(1000)* irá retornar um número entre 0 e 999.

Caso queira um intervalo definido de números, forneça os dois números como parâmetros, dessa forma: *random(50,400)*. Então, *random* irá processar um número aleatório entre o menor

número e o máximo número (-1). Por exemplo, `random(10,100)` irá retornar um número aleatório entre 10 e 99.

No loop principal do programa que vimos primeiro no início e terminar valores RGB e descobrir o que é valor necessário.

No entanto, é necessário um incremento para que haja a mudança de um valor para o outro e utilizar as 256 combinações (com o valor de PWM podemos combinar entre 0 e 255). Fazemos isso assim:

```
for (int x=0; x<3; x++) {  
  INC[x] = (RGB1[x] - RGB2[x]) / 256; }  
}
```

Este loop define os valores pelo incremento nos canais R, G e B por meio do resultado da diferença entre os dois valores de brilho e dividindo-os por 256.

Temos, então, outro laço `for`

```
for (int x=0; x<256; x++) {  
  red = int(RGB1[0]);  
  green = int(RGB1[1]);  
  blue = int(RGB1[2]);  
  analogWrite (RedPin, red);  
  analogWrite (GreenPin, green);  
  analogWrite (BluePin, blue);  
  delay(100);  
  RGB1[0] -= INC[0];  
  RGB1[1] -= INC[1];  
  RGB1[2] -= INC[2];  
}
```

Isso define as cores vermelho, verde e azul para os valores da matriz RGB1, e transfere esses valores para os pinos 9, 10 e 11, depois deduz o valor do incremento, em seguida, repete este processo 256 vezes até desaparecer lentamente a partir de uma cor aleatória para o próximo passo.

O atraso de 100 ms entre cada passo garante uma lenta e constante progressão das combinações de cores. É possível ainda ajustar esse valor para mais lento ou mais rápido ou podemos ainda adicionar um potenciômetro para permitir que um usuário defina a velocidade que desejar.

Depois de ter tomado 256 passos lentos de uma combinação de cor para outra, a matriz RGB1 terá os mesmos valores (quase) como a matriz RGB2. Precisamos agora decidir sobre outro conjunto de três valores aleatórios prontos para a próxima vez. Fazemos isso com outro `for`:

```
for (int x=0; x<3; x++) {  
  RGB2[x] = random(556)-300;  
  RGB2[x] = constrain(RGB2[x], 0, 255);  
  delay(1000);  
}
```

O número aleatório é escolhido num intervalo entre 0 e 556 (256 + 300) e, em seguida é subtraído de 300. A razão pela qual fazemos isso é para tentar produzir cores primárias de tempos em tempos para garantir que nem sempre tenhamos somente tons pastel.

Temos 300 chances fora de 556 de obter um número negativo e, portanto, forçando à uma tendência de termos um ou mais de dois canais de cor. O próximo comando faz certeza de que os números enviados para os pinos PWM não são negativos usando a função `constrain()`.

A função de restringir `constrain()` requer três parâmetros - x, a e b como `constrain(x,a,b)`, onde x é o número queremos restringir, a é a extremidade inferior da gama e b é a extremidade superior. Assim, a função `constrain()` observa o valor de x e garante que ele se encontra dentro do intervalo entre a e b. Se é menor que a, então o define para a, se é maior do que b o define para b. No nosso caso, teremos certeza de que o número está entre 0 e 255, que é o intervalo da nossa saída PWM.

Como usamos números aleatórios [(556) -300] para os nossos valores RGB, alguns desses valores serão menores que zero e a função `constrain()` garante que o valor enviado para o PWM não é menor que zero.




Aplicando uma tendência para uma ou mais cores assegurando tons mais vibrantes e menos cores pastel e assegurando também que ao longo do tempo um ou mais canais estão desligados completamente dando uma mudança mais interessante de luzes (ou humores).

Projeto 9 - Efeito fogo com LED

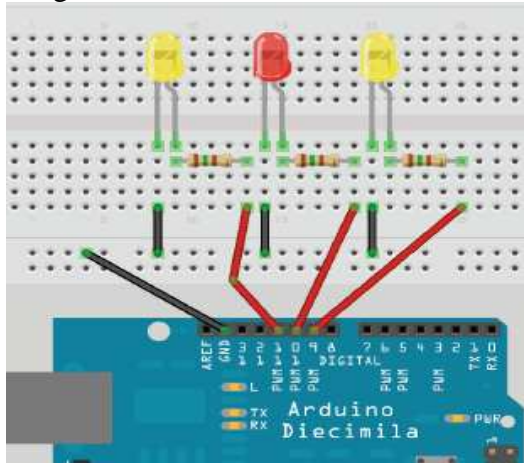
O Projeto 9 vai utilizar LEDs piscando aleatoriamente produzindo um belo efeito, utilizando novamente a saída PWM para recriar o efeito de uma chama.

Se o conjunto for colocado dentro de um ambiente apropriado (uma minilareira, por exemplo), poderia criar um efeito especial de fogo, ou ainda pode-se colocá-lo em uma farsa casa em chamas para dar um efeito de fogo. Este é um simples exemplo de como os LEDs podem ser utilizados para criar efeitos para filmes, peças teatrais, novelas, etc.

O que você precisa

Red Diffused LED	
2 x Yellow Diffused LED's	
3 x 150Ω Resistor	

Montagem



Abra o IDE Arduino e digite o seguinte código:

```
// Projeto 9 - Efeito de LED de fogo
int ledPin1 = 9;
int ledPin2 = 10;
int ledPin3 = 11;
void setup()
{
  pinMode(ledPin1, OUTPUT);
  pinMode(ledPin2, OUTPUT);
  pinMode(ledPin3, OUTPUT);
}
void loop()
{
  analogWrite(ledPin1, random(120)+135);
  analogWrite(ledPin2, random(120)+135);
  analogWrite(ledPin3, random(120)+135);
  delay(random(100));
}
```

Agora pressione o botão Verificar / Compilar no topo da IDE para se certificar de que não há erros em seu código. Se este é bem sucedido agora você pode clicar no botão Upload para carregar o código para o Arduino.

Se tudo estiver certo poderemos ver agora os LEDs em cintilação de uma maneira aleatória para simular uma chama ou efeito de fogo.

Agora vamos dar uma olhada no código e hardware e descobrir como ambos trabalham.

Descrição do código

Primeiro nós declaramos e inicializamos algumas variáveis inteiras que conterão os valores para os pinos digitais que vamos conectar os nossos LEDs, como já foi realizado em projetos anteriores.

```
int ledPin1 = 9;
int ledPin2 = 10;
int ledPin3 = 11;
```

Em seguida, configurá-los para serem saídas.

```
pinMode(ledPin1, OUTPUT);
pinMode(ledPin2, OUTPUT);
pinMode(ledPin3, OUTPUT);
```

O laço principal do programa, em seguida, envia um valor aleatório entre 0 e 120 e, em seguida, adiciona 135 a ele para obter o máximo de brilho dos LEDs, para os pinos de PWM conectados aos números 9, 10 e 11.

```
analogWrite(ledPin1, random(120)+135);
analogWrite(ledPin2, random(120)+135);
analogWrite(ledPin3, random(120)+135);
```

Então, finalmente, temos um atraso aleatório proposital entre ligado e o tempo de 100 ms.

```
delay(random(100));
```

O laço principal, em seguida, começa novamente causando o efeito de luz que se pode ver.

Utilizando um cartão branco onde possa se refletir as luzes ou uma armação como um abajur você poderá ver um efeito de chama muito realista.

Como a construção do *hardware* é bem simples, vamos pular para o próximo Projeto.

Projeto10 – LEDs controlados

Para este projeto utilizaremos o mesmo circuito anterior (Projeto 9), mas agora vamos mergulhar no mundo da comunicação serial e controlar a nossa luz, enviando comandos a partir do PC para o Arduino usando o “*Serial Monitor*” no IDE Arduino, que é uma tela de interface com o hardware do arduino. Este projeto também introduz como manipular sequências de texto. Então deixe a configuração do hardware do mesmo jeito de antes e introduza o novo código:

```
// Projeto 10 - Lampada RGB controlada
char buffer[18];
int red, green, blue;
int RedPin = 11;
int GreenPin = 10;
int BluePin = 9;

void setup()
{
  Serial.begin(9600);
  Serial.flush();
  pinMode(RedPin, OUTPUT);
  pinMode(GreenPin, OUTPUT);
}
```

```

pinMode(BluePin, OUTPUT);
}

void loop()
{
if (Serial.available() > 0) {
int index=0;
delay(100); // Aguarda encher o buffer
int numChar = Serial.available();
if (numChar>15) {
numChar=15;
}
while (numChar-->0) {
buffer[index++] = Serial.read();
}
splitString(buffer);
}
}

void splitString(char* data) {
Serial.print("Data entered: ");
Serial.println(data);
char* parameter;
parameter = strtok (data, " ,");
while (parameter != NULL) {
setLED(parameter);
parameter = strtok (NULL, " ,");
}
// Limpa o texto e os buffers seriais
for (int x=0; x<16; x++) {
buffer[x]='\0';
}
Serial.flush();
}

void setLED(char* data) {
if ((data[0] == 'r') || (data[0] == 'R'))
{
int Ans = strtol(data+1, NULL, 10);
Ans = constrain(Ans,0,255);
analogWrite(RedPin, Ans);
Serial.print("Red is set to: ");
Serial.println(Ans);
}
if ((data[0] == 'g') || (data[0] == 'G'))
{
int Ans = strtol(data+1, NULL, 10);
Ans = constrain(Ans,0,255);
analogWrite(GreenPin, Ans);
Serial.print("Green is set to: ");
Serial.println(Ans);
}
if ((data[0] == 'b') || (data[0] == 'B'))
{
int Ans = strtol(data+1, NULL, 10);
Ans = constrain(Ans,0,255);
analogWrite(BluePin, Ans);
Serial.print("Blue is set to: ");
Serial.println(Ans);
}
}
}

```

Uma vez que você verificou o código, enviá-lo para o seu Arduino.

Agora, quando você carregar o programa nada parece acontecer. Isto porque o programa está esperando por sua entrada. Inicie o “*Serial Monitor*” clicando no respectivo ícone na barra de tarefas no IDE do Arduino.

Na janela de texto “*Serial Monitor*” você pode entrar com os valores de R, G e B para cada um dos três LEDs manualmente e o brilho dos LEDs será alterado e a cor de efeito resultante será a que você programar.

Por exemplo, Se você digitar R255 o LED vermelho irá exibir pleno brilho.

Se você digitar R255, G255, em seguida, os LEDs vermelho e verde irão exibir brilho máximo.

Agora entre com os comandos R127, G100, B255, e você obterá uma cor aparentemente arroxeadada.

Se você digitar, r0, g0, b0 todos os LEDs serão desligados.

O texto de entrada é projetado para aceitar tanto uma letra minúscula ou maiúscula caso de R, G e B e, em seguida, um valor entre 0 e 255. Quaisquer valores acima de 255 serão descartados para no máximo 255. Você pode inserir uma vírgula ou um espaço entre os parâmetros e você pode entrar 1, 2 ou 3 valores de LED em qualquer momento um. Por exemplo,

```
R255 b100, r127 b127 G127, G255 B0,
B127, R0, G255 .
```

Descrição do código

Este projeto apresenta uma série de conceitos novos, incluindo a comunicação serial, ponteiros e manipulação de textos. Então, mantenha-se atento, pois teremos várias explicações.

Primeiro vamos definir uma matriz de *char* (caracteres) para manter nossa cadeia de texto. Fizemos de 18 caracteres, cujos são mais longos do que o máximo de 16, o que nos permitirá garantir que não teremos erros de sobrecargas ao sistema.

```
char buffer[18];
```

Em seguida, iremos configurar os números inteiros para manter os valores das cores vermelho, verde e azul, bem como os valores para os pinos digitais.

```
int red, green, blue;
int RedPin = 11;
int GreenPin = 10;
int BluePin = 9;
```

Na nossa configuração definiremos os três pinos digitais para serem saídas. Mas, antes disso temos o comando *Serial.begin*:

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

```

Serial.flush();
pinMode(RedPin, OUTPUT);
pinMode(GreenPin, OUTPUT);
pinMode(BluePin, OUTPUT);
}

```

Serial.begin diz ao Arduino para começar a comunicação serial e o número dentro dos parêntesis, neste caso 9600, define a taxa de transmissão (caracteres por segundo) que a linha serial irá comunicar.

Obs.: Você deve alterar a taxa de transmissão no *Serial Monitor* para 9600 baud.

O comando *Serial.flush* irá limpar qualquer caractere indevido na linha de série e deixá-lo vazio e pronto para a entrada/saída de comandos.

A linha de comunicação serial é simplesmente uma forma de o Arduino se comunicar com o mundo exterior, neste caso através do PC e do *IDE Arduino Serial Monitor* para o hardware construído.

No circuito principal, temos uma instrução *if*. A condição é verificar

```

if (Serial.available() > 0) {

```

O comando *Serial.available* verifica se caracteres foram enviados para a linha do monitor serial. Se qualquer caractere foi recebido, em seguida, as condições e declarações do código dentro do bloco *if* é executada.

```

if (Serial.available() > 0) {
int index=0;
delay(100); // let the buffer fill up
int numChar = Serial.available();
if (numChar>15) {
numChar=15;
}
while (numChar-->0) {
buffer[index++] = Serial.read();
}
splitString(buffer);
}
}

```

Um inteiro chamado *index* é declarado e inicializado como zero. Este inteiro vai manter a posição de um ponto para os caracteres dentro da matriz de char, como uma referência.

Em seguida, define-se um atraso de 100. O propósito disto é assegurar que o *serial buffer* (o lugar na memória onde os dados são recebidos e armazenados antes do processamento) está cheio antes de continuar o processo com os dados. Se não fizer isso, é possível que a função execute e comece a processar o comando enviado, antes de termos enviado todos os dados. A linha de comunicação serial é muito lenta em comparação com a velocidade do resto do código de execução. Quando você envia uma sequência de dados, a

função *Serial.available* terá imediatamente um valor superior a zero e a função *if* começará a ser executada. Se não houvesse o atraso, o código dentro do *if* começaria a ser executado antes de toda a informação de texto ter sido recebida e o dados seriais poderiam ser apenas os primeiros caracteres de texto inserido.

Depois de ter esperado 100 ms, para preencher o *buffer serial* com os dados enviados, nós então declaramos e inicializamos o inteiro *numChar* para ser o número de caracteres dentro do texto.

Por exemplo, se enviamos este texto para o monitor: R255, G255, B255.

Em seguida, o valor de *numChar* é 17 e não 16, como no final de cada linha de texto, onde há um caractere invisível chamado de "NULL". Este é um símbolo de vazio e simplesmente diz ao Arduino que o fim da linha de texto foi atingido.

A próxima instrução *if* verifica se o valor de *numChar* é superior a 15 ou não e se assim se define como sendo de 15. Isso garante que nós não estouraremos o array *char buffer [18]*;

Depois vem um comando *while*. É um comando que nós ainda não nos deparamos antes, então segue a explanação.

Nós já usamos o laço *for*, que irá percorrer um determinado número de vezes numa sequência. A instrução *while* é também um laço, mas que executa somente enquanto uma condição é verdadeira.

A sintaxe é:

```

while(expression) {
// statement(s)
}

```

Em nosso código ficou como:

```

while (numChar-->0) {
buffer[index++] = Serial.read();
}

```

A condição é verificada simplesmente através do *numChar*, portanto, em outras palavras, é verificar se o valor armazenado no inteiro *numChar* não é zero. Isto é conhecido como um pós-decremento. Em outras palavras, o valor é decrementado depois que é usado. No nosso caso, o loop *while* verifica o valor de *numChar* e, em seguida, subtrai uma unidade a partir dele. Se o valor de *numChar* não era zero antes do decremento, então executa o código. Está definido para o comprimento da sequência de texto que entraram na janela do Monitor Serial. Assim, o código dentro do loop *while* irá ser executado muitas vezes. O código dentro do laço *while* é:

```
buffer[index++] = Serial.read();
```

Que define cada elemento da matriz tampão para cada caractere lido a partir da linha de série. Em outras palavras, ele enche o buffer array com os dados que inserimos na janela do Serial Monitor. O comando `Serial.read()` lê a série de dados de entrada, um byte de cada vez.

Portanto, agora que a nossa matriz de caracteres foi preenchida com os dados que entraram no Serial Monitor, o `while` terminará assim que `numChar` chegar a zero. Depois temos:

```
splitString(buffer);
```

Que é uma chamada para uma das duas funções que criamos e chamamos de `splitString()`. Essa função é descrita a seguir:

```
void splitString(char* data) {
  Serial.print("Data entered: ");
  Serial.println(data);
  char* parameter;
  parameter = strtok (data, " ,");
  while (parameter != NULL) {
    setLED(parameter);
    parameter = strtok (NULL, " ,");
  }
  for (int x=0; x<16; x++) {
    buffer[x]='\0';
  }
  Serial.flush();
}
```

Podemos ver que a função não retorna nenhum tipo de dado, portanto podemos ver que seu tipo foi criado para anular possíveis continuidades. Passamos o parâmetro de uma função que é um tipo de `char` (caractere) e que trataremos como de dados. No entanto, nas linguagens de programação C e C++ você não tem permissão para enviar uma matriz de caracteres para uma função. Temos que contornar isso, usando um parâmetro. Nosso parâmetro será um asterisco (*) e foi adicionada ao (*) o nome de dados variáveis. Parâmetros são bastante avançados no C de modo que não vamos entrar em muitos detalhes sobre eles. Tudo que você precisa saber agora é que por declaração de dados como um parâmetro, simplesmente uma variável aponta para outra variável.

Você pode apontá-lo para o endereço que a variável é armazenada na memória usando o símbolo `&`, ou em nosso caso, para o valor armazenado a esse endereço de memória usando o símbolo `*`. Usaremos isso para enganar o sistema, como nós não temos permissão para enviar um caractere à matriz para uma função. No entanto, estamos autorizados a enviar um indicador para uma matriz de caracteres para a nossa função.

Assim, nós declaramos uma variável do tipo de dados `char` e chamamos de dados, mas o símbolo `*`, isso significa que está apontando para o valor armazenado dentro do `buffer`.

Quando chamamos `splitString` enviamos para o seu conteúdo o `buffer` (na verdade o parâmetro para ele, como vimos acima).

```
splitString(buffer);
```

Então nós chamamos a função e passamos para o seu conteúdo a matriz de caracteres do `buffer`.

O primeiro comando é

```
Serial.print("Data entered: ");
```

e esta é a nossa forma de enviar dados de volta do Arduino para o PC. Neste caso, o comando de impressão envia o que está dentro do parêntese para o PC, através do cabo USB, onde podemos ler no Serial Monitor.

Neste caso, já enviamos as palavras "Data entered:". O texto deve ser colocado entre aspas ". A linha seguinte é semelhante

```
Serial.println(data);
```

E mais uma vez, enviou dados para o PC. Por exemplo:

```
R255 G127 B56
```

Em seguida,

```
Serial.println(data);
```

Este comando irá enviar essa sequência de texto para o PC e imprimi-lo na janela do Serial Monitor, se algo for digitado na janela do Serial.

Desta vez, o comando de impressão tem "ln" no fim torná-lo `println`. Isto simplesmente significa impressão/imprimir na linha do Serial Monitor.

Quando imprimir utilizando o comando `print`, o cursor (o ponto onde o símbolo irá aparecer) permanece no final de tudo o que tenha imprimido.

Quando usamos o comando `println` um comando de avanço de linha é emitido ou em outras palavras, as impressões de texto e, em seguida, o cursor cai para a linha seguinte.

```
Serial.print("Data entered: ");
Serial.println(data);
```

Se olharmos para os nossos dois comandos de impressão, o primeiro imprime "Os dados inseridos:" e então o cursor permanece no final do

texto. A próxima impressão vai imprimir os dados, ou em outras palavras, o conteúdo da matriz chamado tampão e depois emitir uma linha de alimentação, ou deixar cair o cursor para a próxima linha. Isto significa que, se emitir outra impressão ou declaração `println` após isso tudo que é impresso na janela do Serial Monitor será exibido na próxima linha por baixo da última.

Em seguida, criamos um novo tipo de dados `char` chamado parâmetro.

```
Char* parameter;
```

e como vamos usar essa variável para acessar elementos da matriz de dados, ele deve ser do mesmo tipo, portanto o símbolo `*`. Você não pode passar dados de um tipo de variável para outro tipo de variável, os dados devem ser convertidos em primeiro lugar. Esta variável é outro exemplo de alguém que tem o escopo local. Pode ser visto apenas pelo código dentro dessa função. Se você tentar acessá-la fora parâmetro da variável função `splitString` você obterá um erro.

Em seguida, usamos um comando `strtok`, que é um comando muito útil para nos capacitar para manipular textos de comando. `strtok` é a junção do nome de *string e Token* e o seu objetivo é dividir um texto usando símbolos. No nosso caso o *símbolo* que estamos procurando é um espaço ou uma vírgula. Ele é usado para dividir sequências de caracteres de texto em partes menores e legíveis ao programa. Passamos a matriz de dados para o comando `strtok` como o primeiro argumento e os símbolos (incluído o dentro das aspas) como o segundo argumento. Por isso:

```
parameter = strtok(data, " ,");
```

E ele divide a sequência nesse ponto. Portanto, estamos a usá-lo para ajustar o parâmetro a ser a parte da cadeia até um espaço ou uma vírgula. Então, se a nossa cadeia de texto foi

R127 G56 B98

Então, após esta declaração o valor do parâmetro será

R127

O comando `strtok` simplesmente divide a sequência digitada para a primeira ocorrência de um espaço ou de uma vírgula.

Depois de ter definido o parâmetro variável para a parte da cadeia de texto que queremos retirar (ou seja, o bit do primeiro espaço ou vírgula) em

seguida, daremos um comando de fim de comando.

```
while(parameter != NULL) {
```

Dentro do loop que chamamos de nossa segunda função

```
setLED(parameter);
```

Que veremos mais tarde. Em seguida, ele define o parâmetro variável para a próxima parte da sequência para o próximo espaço ou vírgula. Fazemos isso passando para `strtok` um parâmetro `NULL`

```
parameter = strtok(NULL, " ,");
```

Isso informa ao programa para continuar de onde parou.

Então, temos toda essa parte da função:

```
char* parameter;
parameter = strtok(data, " ,");
while (parameter != NULL) {
    setLED(parameter);
    parameter = strtok(NULL, " ,");
}
```

é simplesmente a cadeia de comandos que permite ajustar a próxima função chamada `setLED()`.

A parte final desta função simplesmente preenche o buffer com conjunto dos caracteres, que é feito com o `"/0"` símbolo que nivela e em seguida, os dados de série estão prontos para o próximo conjunto de dados que poderão ser inseridos.

```
// Clear the text and serial buffers
for (int x=0; x<16; x++) {
    buffer[x]='\0';
}
Serial.flush();
```

A função `setLED()` vai identificar cada parte da sequência de texto e definir o LED correspondente à cor que se escolher. Assim, se a sequência de texto é:

G125 B55

A função `splitString()` divide o texto em dois componentes separados

G125
B55

e envia essa sequência de texto abreviado para o `setLED()` da função, que vai lê-lo, definir o LED que escolhido e configurá-lo para o brilho correspondente do valor.

Então vamos dar uma olhada na segunda função chamada `setLED()`:

```
void setLED(char* data) {
  if ((data[0] == 'r') || (data[0] == 'R'))
  {
    int Ans = strtol(data+1, NULL, 10);
    Ans = constrain(Ans,0,255);
    analogWrite(RedPin, Ans);
    Serial.print("Red is set to: ");
    Serial.println(Ans);
  }
  if ((data[0] == 'g') || (data[0] == 'G'))
  {
    int Ans = strtol(data+1, NULL, 10);
    Ans = constrain(Ans,0,255);
    analogWrite(GreenPin, Ans);
    Serial.print("Green is set to: ");
    Serial.println(Ans);
  }
  if ((data[0] == 'b') || (data[0] == 'B'))
  {
    int Ans = strtol(data+1, NULL, 10);
    Ans = constrain(Ans,0,255);
    analogWrite(BluePin, Ans);
    Serial.print("Blue is set to: ");
    Serial.println(Ans);
  }
}
```

Podemos ver que esta função contém três comandos muito semelhantes de `if`. Por isso vamos dar uma olhada em apenas um deles, pois os outros dois são quase idênticos.

```
if ((data[0] == 'r') || (data[0] == 'R')) {
  int Ans = strtol(data+1, NULL, 10);
  Ans = constrain(Ans,0,255);
  analogWrite(RedPin, Ans);
  Serial.print("Red is set to: ");
  Serial.println(Ans);
}
```

A instrução `if` verifica se o primeiro caractere no campo de dados da sequência (`data[0]`) é uma letra `r` ou `R` (maiúscula ou minúsculas são totalmente diferentes para a linguagem em C). Usamos o comando lógico OR cujo símbolo é `||` para verificar se a letra é maiúscula ou minúscula. Sabendo que é um `r` ou um `R`, a instrução `if` irá agora dar o comando de brilho do LED vermelho e assim executa o código dentro da sequência.

Primeiro nós declaramos um inteiro chamado `Ans` (que tem valor temporário) e usamos o comando `strtol` para converter os caracteres após a letra digitada inicialmente, sendo ela `r` ou `R`, para valores inteiros.

O comando `strtol` tem três parâmetros que são levados em consideração; estes são: a sequência que estamos utilizando, um ponto após o caractere

inteiro (`NULL`) e, em seguida, a base que, no nosso caso é base igual a 10, como estamos usando números decimais normais (ao contrário de binário, octal ou hexadecimal que seriam de base 2, 8 e 16, respectivamente). Assim, em outras palavras, declaramos uma série de comandos que identificarão um inteiro e irão configurá-lo para o valor de brilho do led escolhido.

Usamos o comando `constrain` (restringir) para certificar-se que o número em `Ans` vai de 0 a 255 e não mais que isso. Em seguida, o comando `analogWrite` envia para o pino de saída e o valor de `Ans`. As outras duas declarações `if` executam comandos idênticos ao explicado.

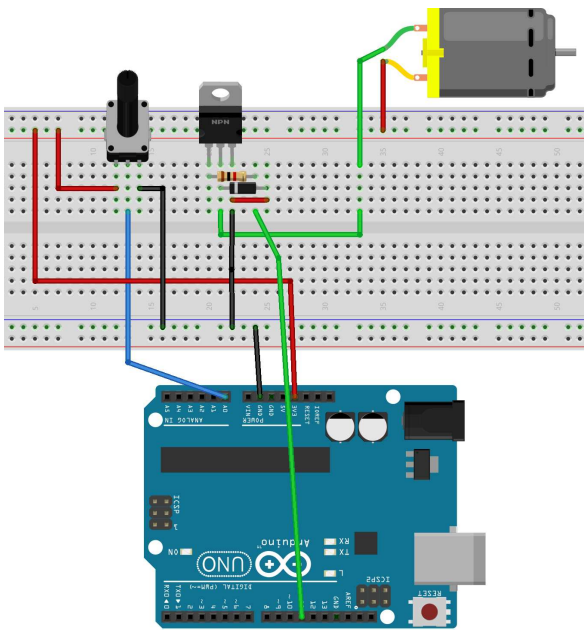
É necessário que se estude mais a fundo os comandos citados até aqui, pois têm diversas outras aplicações e funções diferentes, além das utilizadas até agora. Tivemos uma introdução básica de novos conceitos nestes projetos.

Projeto 11 – Acionamento de um motor CC

Este projeto trabalhará com acessórios diferentes, como fontes, motores, diodos, etc.

O que você precisa

DC Motor	
4K7 Potentiometer	
TIP-120 Transistor	
1N4001 Diode	
1KΩ Resistor	
9v Power Supply	



```
int potPin = 0; // Analog in 0
connected to the potentiometer
int transistorPin = 11
; // connected to the base of the
transistor
int potValue = 0; // value returned
from the potentiometer
void setup() {
// set the transistor pin as output:
pinMode(transistorPin, OUTPUT);
}
void loop() {
// read the potentiometer, convert it
to 0 - 255:
potValue = analogRead(potPin) / 4;
// use that to control the
transistor:
analogWrite(transistorPin, potValue);
}
```

Antes de ligar o circuito, verifique se tudo foi conectado corretamente e em especial o diodo que tem polarização. Se ligá-lo de maneira errada, isso pode resultar em danos permanentes ao o Arduino e componentes. Quando estiver certo que as ligações estão corretas, faça o upload do seu código.

```
int potPin = 0;
int transistorPin = 11;
int potValue = 0;
void setup() {
pinMode(transistorPin, OUTPUT);
}
void loop() {
potValue = analogRead(potPin) / 4;
analogWrite(transistorPin, potValue);
}
```

Este código é muito simples. Declaramos 3 inteiros que armazenam os valores dos pinos que se ligam ao nosso potenciômetro, ao pino que liga

ao transistor e ao valor lido a partir do potenciômetro.

Na função setup () vamos configurar o *pinMode* do pino do transistor para saída.

O *potValue* do loop principal está definido para ler o valor lido a partir do pino analógico 0 (*potPin*) e, em seguida, é dividido por 4.

Temos que dividir o valor lido por 4, para fazer a relação com o valor analógico que irá variar de 0 a 0 volts a 1023 para 5 volts. O valor do pino do transistor só pode variar de 0-255, de modo que dividindo o valor do pino analógico 0 (max 1023) por 4 obtemos o valor máximo de 255 para a fixação do pino digital 11 (usando *analogWrite* por isso estamos usando PWM).

O código, em seguida, escreve para o pino do transistor o valor ajustado no potenciômetro. Em outras palavras, quando você gira a potenciômetro, diferentes valores são lidos (variam de 0 a 1023) e estes são convertidos para a faixa de 0 a 255 e, em seguida, esse valor é escrito (via PWM) para o pino digital 11 que muda a velocidade do motor de corrente contínua. Gire o potenciômetro todo para a esquerda e o motor desliga, vire para a direita e ele acelera até que atinja a velocidade máxima.

Visão geral do hardware

O hardware para o Projeto 11 é como abaixo:

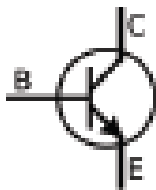
DC Motor	
4K7 Potentiometer	
TIP-120 Transistor	
1N4001 Diode	
1KΩ Resistor	
9v Power Supply	

O circuito é essencialmente dividido em duas seções: (a) A seção 1 é o nosso potenciômetro, que está ligado à +3,3 V e ao terra com o pino central de entrada no pino analógico A0. Como o potenciômetro é giratório, a variação de resistência permite tensões de 0 a 3,3 V, onde o valor é lido usando o pino analógico A0. (b) A

segunda seção é a que controla o motor. Os pinos digitais do Arduino fornecem um máximo de 40 mA. O motor requer em torno de 500 mA para funcionar em alta velocidade e isto é, obviamente muito alto para o Arduino. Se fôssemos tentar conduzir o motor diretamente a partir de um pino do Arduino um grave e permanente dano poderia ocorrer. Portanto, precisamos encontrar uma maneira de fazê-lo de maneira indireta.

Este projeto controla a velocidade do motor, por isso precisamos de uma maneira de controlar essa tensão para acelerar ou retardar o motor. Este é o lugar onde o transistor TIP-120 atua.

Um transistor é essencialmente um comutador digital. Ele também pode ser utilizado como um amplificador de potência. O símbolo eletrônico para um transistor bipolar é mostrado a seguir:



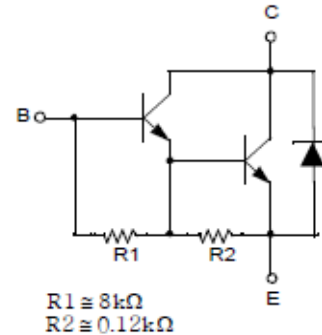
O transistor tem três pernas, uma é a base, outra é o coletor e a terceira é o emissor. Estes são identificados como B, C e E na figura abaixo.



No nosso circuito, temos 3,3 volts indo para o coletor através do motor. A base é ligada através de um resistor de 1 kΩ ao pino digital 11. O emissor é ligado ao pino terra. Enviamos pulsos via PWM ao pino 11, e esta tensão é reduzida usando um resistor de 1 kΩ. Sempre que se aplica uma tensão à base, através do pino 11, o que permite que corrente flua através do coletor ao emissor e, portanto, acionar o motor, que está ligado em série com este circuito.

Um motor é um eletroímã que tem um campo magnético induzido enquanto uma tensão é fornecida. Quando a energia é removida, o colapso do campo magnético pode produzir uma tensão reversa. Isso pode danificar seriamente o seu Arduino e é por isso que o diodo foi introduzido no circuito. A listra branca marcada

no diodo indica o lado que se liga no terra. Ele filtra o fluxo a partir do lado positivo para o lado negativo. O diodo irá funcionar como uma válvula para impedir refluxo de corrente. O diodo em nosso circuito é, portanto, inserido para proteger o Arduino.



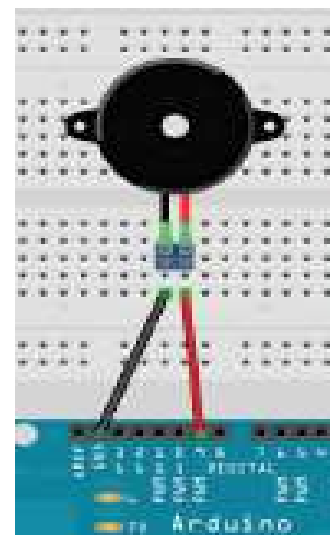
Circuito equivalente do transistor TIP120.

Projeto 12 – Atuador piezelétrico

Neste projeto vamos usar um simples circuito para produzir sons usando um Arduino e uma sirene piezelétrica.

O que você precisa

Piezo Disc	
Terminal Block	



Montagem

Quando você executar esse código o Arduino fará soar um som muito semelhante aos irritantes cartões de aniversário que você pode comprar em qualquer banca de jornais e que quando são

abertos tocam uma música. Vamos dar uma olhada neste código e ver como funciona e descobrir o que é um disco piezo.

Descrição do código

```
// Projeto 12 - Atuador piezeletrico
int speakerPin = 9;
int length = 15; // Numero de notas
char notes[] = "ccggaagffeeddc "; //
Espaco representa uma pausa
int beats[] = { 1, 1, 1, 1, 1, 1, 1, 2, 1,
1, 1, 1, 1, 1, 2, 4 };
int tempo = 300;

void playTone(int tone, int duration) {
for (long i=0; i < duration*1000L;
i+=tone*2) {
    digitalWrite(speakerPin, HIGH);
    delayMicroseconds(tone);
    digitalWrite(speakerPin, LOW);
    delayMicroseconds(tone);
}
}

void playNote(char note, int duration) {
char names[] = { 'c', 'd', 'e', 'f', 'g',
'a', 'b', 'C' };
int tones[] = { 1915, 1700, 1519, 1432,
1275, 1136, 1014, 956 };
// Toca o tom corresponde ao nome da nota
for (int i = 0; i < 8; i++) {
    if (names[i] == note) {
        playTone(tones[i], duration);
    }
}
}

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

void loop() {
for (int i = 0; i < length; i++) {
    if (notes[i] == ' ') {
        delay(beats[i] * tempo); // rest
    } else {
        playNote(notes[i], beats[i] * tempo);
    }
    // Pausa entre notas
    delay(tempo / 2);
}
}
```

Neste projeto estamos fazendo sons usando um disco piezelétrico. Um disco piezo pode fazer nada mais do que uma vibração quando se aplica uma tensão ao mesmo. Assim, para obter os tons que podemos ouvir, precisamos fazer várias vibrações e muitas vezes num instante rápido o suficiente para que ele realize uma nota reconhecível e que seja audível.

O programa começa por estabelecer as variáveis necessárias. No piezo de sirenes, o cabo vermelho é positivo e é ligado ao pino 9 do Arduino.

```
int speakerPin = 9;
```

A música que vamos tocar é composta de 15 notas.

```
int length = 15; // the number of notes
```

As notas da melodia são armazenadas em uma matriz de caracteres como um texto.

```
char notes[] = "ccggaagffeeddc ";
```

Outra matriz, desta vez de números inteiros, é configurada para armazenar o comprimento de cada nota.

```
int beats[] = { 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1,
1, 1, 2, 4 };
```

E, finalmente, definir um tempo para a música a ser tocada:

```
int tempo = 300;
```

Em seguida, você vai perceber que nós declaramos duas funções antes do nosso setup() e loop(). Não importa se colocarmos nossas próprias funções, antes ou depois setup () e loop (). Quando o programa é executado, o código dentro dessas duas funções não será executado antes do setup () pois ainda não chamamos essas funções ainda.

Vamos olhar para as funções de configuração e loop antes de olharmos para o playTone e funções playNote. Tudo o que acontece no setup () é que atribuir o alto-falante no pino (9) como uma saída.

```
void setup() {
    pinMode(speakerPin, OUTPUT);
}
```

No loop principal do programa, temos um if / else declaração dentro de um loop for.

```
for (int i = 0; i < length; i++) {
    if (notes[i] == ' ') {
        delay(beats[i] * tempo); // rest
    } else {
        playNote(notes[i], beats[i] * tempo);
    }
}
```

Como você pode ver, a primeira instrução “if ” tem como condição, que o elemento da matriz [i] contém um caractere de espaço.

```
if (notes[i] == ' ')
```

Se for verdadeiro, então o código de dentro do seu bloco é executado.

```
delay(beats[i] * tempo); // rest
```

e isso simplesmente provoca um atraso nas notas. Temos, então, outra declaração.

```
else {  
    playNote(notes[i], beats[i] * tempo);  
}
```

Depois de uma afirmação se pode estendê-lo como outra declaração. Um else é executado se o condição dentro do if é falsa. Assim, por exemplo, vamos dizer que tivemos um teste inteiro e seu valor era de 10 e esta declaração if / else:

```
if (test == 10) {  
    digitalWrite(ledPin, HIGH)  
} else {  
    digitalWrite(ledPin, LOW)  
}
```

Então se o teste teve um valor de 10 o ledPin seria definido como Alto. Se o valor do teste foi diferente de 10, o código dentro da outra declaração seria realizada e em vez disso o ledPin seria definido como Baixo.

A declaração adiante chama uma função chamada *playNote* e passa dois parâmetros. O primeiro parâmetro é o valor das notas [i] e o segundo é o valor calculado a partir de batidas [i] ritmo *.

```
playNote(notes[i], beats[i] * tempo);
```

Após instrução if /else realizada, há um atraso cujo valor é calculado dividindo ritmo por 2.

```
delay(tempo / 2);
```

Vamos agora dar uma olhada nas duas funções que criamos para este projeto.

A primeira função que é chamada a partir do programa principal loop é *playNote*.

```
void playNote(char note, int duration) {  
    char names[] = { 'c', 'd', 'e', 'f', 'g',  
                    'a',  
                    'b', 'c' };  
    int tones[] = { 1915, 1700, 1519, 1432,  
                   1275,  
                   1136, 1014, 956 };  
    // play the tone corresponding to the  
    note name  
    for (int i = 0; i < 8; i++) {  
        if (names[i] == note) {  
            playTone(tones[i], duration);  
        }  
    }  
}
```

Dois parâmetros foram passados para a função e dentro da função à estes foram dados os nomes das notas (names - caracteres) e duração (duration-inteiro).

A função cria uma matriz local variável de dado. Esta variável tem locais de escopo que só é visível para essa função e não fora dela.

Em seguida, criar outra matriz de dados do tipo inteiro e esta matriz armazena os números que correspondem à frequência dos tons, em kilohertz, de cada uma das notas nos nomes da matriz.

```
int tones[] = { 1915, 1700, 1519, 1432, 1275,  
               1136, 1014, 956 };
```

Depois de configurar as duas matrizes existe um loop que olha através das 8 notas e compara com a nota enviada para a função.

```
for (int i = 0; i < 8; i++) {  
    if (names[i] == note) {  
        playTone(tones[i], duration);  
    }  
}
```

A música que é enviada para este função é "Ccggaagffeeddc" para a primeira nota será um meio C.

O loop for compara a nota com as notas da matriz e se houver uma correspondência, chama a segunda função, chamada *playTone*, para jogar o tom correspondente.

A segunda função é chamado *playTone*.

```
void playTone(int tone, int duration)  
{  
    for (long i = 0; i < duration *  
         1000L; i +=  
         tone * 2) {  
        digitalWrite(speakerPin, HIGH);  
        delayMicroseconds(tone);  
        digitalWrite(speakerPin, LOW);  
        delayMicroseconds(tone);  
    }  
}
```

Dois parâmetros são passados para esta função. O primeiro é o tom (em quilohertz) que queremos que o piezo altofalante reproduza e a segunda é a duração (composto por cálculo através da função *beats [i] * ritmo*). A função inicia um loop

```
for (long i = 0; i < duration *  
     1000L; i += tone * 2)
```

À medida que cada loop deve ter de um comprimento diferente para fazer com que cada nota tenha a mesma duração (como o atraso difere

entre vibrações para produzir a frequência desejada) o loop será executado e a duração multiplicada por 1000 e o incremento do loop é o valor de tonalidade multiplicado por dois.

Dentro do loop podemos observar que simplesmente o pino conectado ao piezo-elétrico tem uma ordem de sinal de saída ALTO, espera-se um curto período de tempo, em seguida, é enviado um sinal BAIXO, então espera-se mais um curto período de tempo, depois o processo é retomado.

```
digitalWrite(speakerPin, HIGH);  
delayMicroseconds(tone);  
digitalWrite(speakerPin, LOW);  
delayMicroseconds(tone);
```

Esses cliques repetitivos, de cumprimentos diferentes e com diferentes pausas (de microssegundos apenas) entre os intervalos, faz com que o piezo produza um som gerado a partir de diferentes frequências.

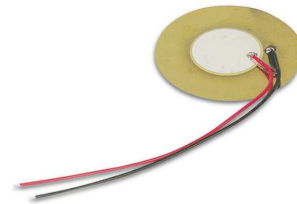
Visão geral do hardware

A única parte de hardware utilizada neste projeto é uma sirene piezoelétrica ou piezoelétrica. Este dispositivo simples é constituído por uma fina camada de cerâmica ligada a um disco metálico. Materiais piezoelétricos, são alguns cristais e cerâmica, que têm a habilidade de produzir eletricidade a partir da tensão (*stress*) mecânica quando aplicado a eles.

Têm aplicações, tais como a produção e detecção de som, a geração de elevada tensões, geração de frequência eletrônica, microbalanças e trabalho em ultrafinas de focagem ópticas.

O efeito também é reversível, em que, se um campo elétrico é aplicada através do material piezoelétrico irá fazer com que o material mude de forma (por menos que seja 0,1% em alguns casos).

Para produzir os sons de um disco piezo, um campo elétrico é ligado e desligado muito rápido, para fazer o material mudar de forma e, portanto, causar uma vibração (como um tambor pequeno). Por mudar a frequência dos pulsos, o disco deforma-se centenas ou milhares de vezes por segundo e, conseqüentemente, causando o som. Ao alterar a frequência das vibrações e o tempo entre eles, notas específicas podem ser produzidas.



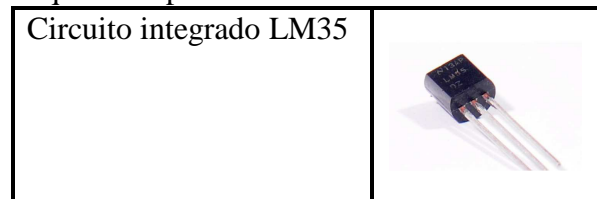
Exercícios

1. Alterar as notas e batidas para fazer outras músicas, como feliz Aniversário ou Feliz Natal.
2. Escreva um programa para fazer um tom subindo e descendo do piezo, semelhante a um alarme de carro ou sirene da polícia.

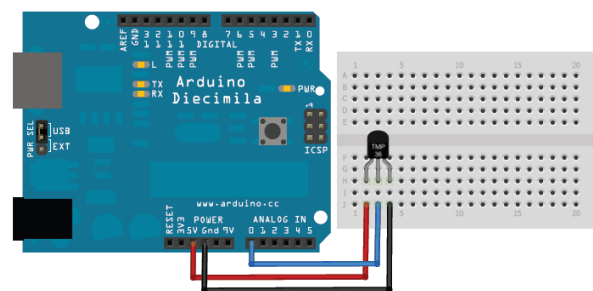
Projeto 13 – Sensor de temperatura com o CI LM35

Neste projeto mostraremos como construir um sensor de temperatura usando o circuito integrado LM35.

O que você precisa



Montagem:



Um sensor de temperatura embutido no CI LM35 lê uma resistência elétrica, que é processada no CI e convertida no valor de temperatura em °C.

O CI LM35 é um circuito integrado que é alimentado com tensão de +5 Vcc. Esta tensão é adequada para o Arduino já que ele possui um pino de saída de 5 V. O CI possui três terminais: dois para alimentação de 5 V e um terceiro do sinal de saída analógico.

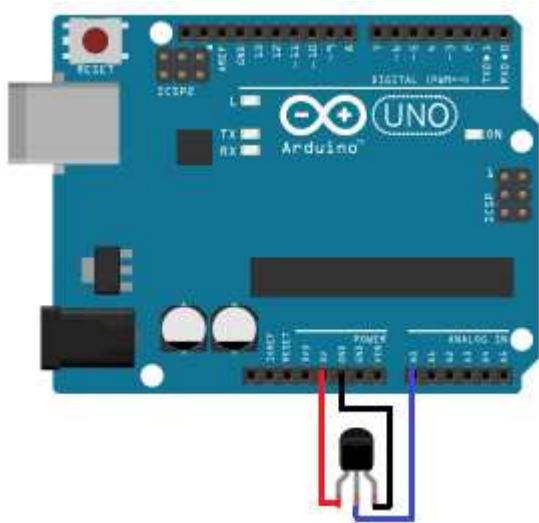
O terminal de saída analógica fornece uma tensão que é diretamente proporcional à temperatura em °C. De acordo com a Figura abaixo, o terminal 2

fornece uma saída de 1 mV/0,1°C (ou 10 mV/°C). Assim, para determinar a temperatura em °C deve-se ler a tensão em mV do terminal 2 e dividir por 10 para ter o valor de temperatura em °C. Por exemplo, se o terminal 2 fornece uma leitura de tensão de 315 mV corresponderá a temperatura de 31,5°C.

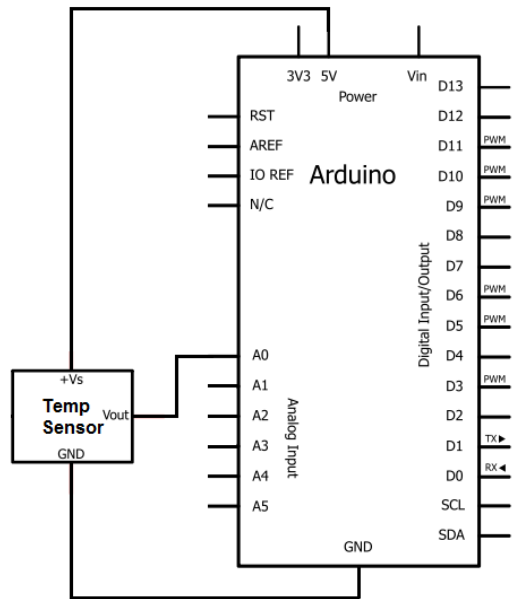


Visão do hardware

O terminal 1 do CI recebe a tensão +5 V e o terminal 3 é aterrado no pino GND do Arduino. O terminal 2 é conectado ao pino A0 de entrada analógica do Arduino, conforme mostra a Figura abaixo.



O mesmo circuito pode ser esquematizado como segue.



Projeto 13 – Descrição do código

```
// Define o pino de saída do CI LM35
int outpin= 0;

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

// Laço principal
void loop()
{
  int rawvolt= analogRead(outpin);
  float milivolt= (rawvolt/1024.0) * 5000;
  float celsius= milivolt/10;
  Serial.print(celsius);
  Serial.print(" graus Celsius, ");
  delay(5000);
}
```

O sinal de tensão de saída do CI LM35 é lido na variável rawvolt. Este valor é proporcional à tensão de 5000 mV, dividido por 1024 (que é o máximo valor inteiro lido pela porta analógica do Arduino). O valor resultante de tensão (em mV) é gravado na variável milivolt. Por último, a temperatura em °C é calculada dividindo milivolt por 10 (fator de conversão de tensão em mV para temperatura °C).

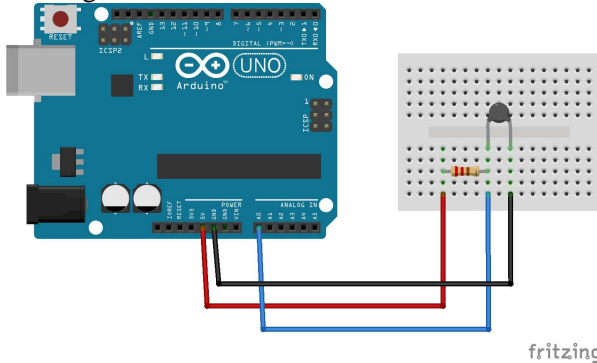
No final do *sketch* é colocado um atraso de 5 s para leitura da temperatura do sensor LM35. Este valor pode ser ajustado conforme a necessidade.

Projeto 14 - Sensor de temperatura Termistor NTC 10 kΩ

Neste projeto mostraremos como construir um sensor de temperatura usando o termistor NTC de 10 kΩ.

O que você precisa
Termistor NTC 10 kΩ

Montagem



Termistores

Termistores são resistores termicamente sensíveis. Os termistores são muito utilizados em aplicações que necessitem de uma alta sensibilidade com mudança à temperatura, pois eles são extremamente sensíveis a mudanças relativamente pequenas de temperaturas.

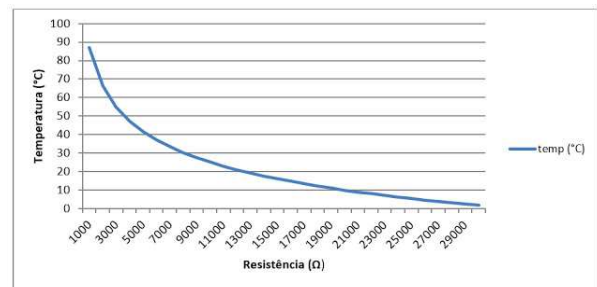
Todavia não são lineares. A resposta da variação de temperatura não é linear à variação de sua resistência, mas pode-se obter a relação entre a resistência e a temperatura para faixas pequenas de variação de temperatura através da equação de Steinhart-Hart.

Existem dois tipos de termistores: o **PTC** (Coeficiente Positivo de Temperatura) e o **NTC** (Coeficiente Negativo de Temperatura), essa distinção é devida ao material de sua construção. Esses dispositivos são construídos a partir de misturas de óxidos semicondutores, como titanato de bário para os PTCs e óxidos de magnésio, níquel, cobalto, titânio, ferro e cobre para os NTCs.

Os termistores NTC são sensíveis a variações de temperatura. Sua resistência diminui à medida que a temperatura aumenta. Desenvolvidos com uma tecnologia que permite tolerância de variação de 1%.

São usados principalmente para faixas de medições entre -55° a 150°C. Devido ao baixo custo e ao excelente desempenho, o termistor NTC possui uma ampla utilização nas indústrias, seja para refrigeração, condicionamento de ar, em automóveis, controle de temperatura, sistemas de detecção e alarmes contra incêndio, dentre outras. A curva que define o comportamento da resistência em função da temperatura tem um

comportamento exponencial, como pode ser visto no gráfico abaixo.



Tensão de saída do circuito

Para este circuito divisor de tensão determina-se a equação para o sinal de saída injetado ao microcontrolador da seguinte maneira:

$$I = \frac{V_{cc}}{R1 + RNTC}$$

$$I = \frac{VR1}{R1}$$

$$\frac{V_{cc}}{R1 + RNTC} = \frac{VR1}{R1}$$

$$\frac{V_{cc}}{R1 + RNTC} = \frac{VR1}{R1}$$

$$VR1 = \frac{V_{cc} \times R1}{R1 + RNTC}$$

Como a tensão de saída $V_{out} = VR1$, temos:

$$V_{out} = \frac{V_{cc} \times R1}{R1 + RNTC}$$

Como V_{out} do circuito é igual à V_{in} da entrada analógica do Arduino, pode-se adaptar a equação e inseri-la ao código do programa para determinar corretamente o valor da resistência do NTC. Este valor de resistência será utilizado na equação de Steinhart-Hart:

$$V_{out}(\text{Divisor}) = V_{in}(\text{PIC})$$

$$RNTC = \left(\frac{V_{cc} + R1}{V_{in}(\text{PIC})} \right) - R1$$

Equação de Steinhart-Hart

Analisando o gráfico da curva resistiva característica de um termistor NTC 10 kΩ, observa-se que a resposta do sensor à variação da temperatura medida não é linear, desta forma, a

interpretação do sinal de entrada injetado ao microcontrolador precisar ser tratada através de uma equação que torne precisa qualquer temperatura medida.

A relação entre resistência e temperatura no NTC é dada pela equação de Steinhart-Hart:

$$T (^{\circ}C) = \frac{1}{a + b \times \ln(RNTC) + c \times \ln^3(RNTC)}$$

Para utilizá-la, selecionam-se as constantes a, b e c definidas no manual do fabricante do Termistor NTC ou através de medições realizadas em ensaio quando estas informações não estiverem disponíveis.

As constantes para o sensor do kit são:

a = 0,0011303

b = 0,0002339

c = 0,00000008863

Basta agora aplicar estes valores à fórmula inserida no programa do microcontrolador para determinar a correta leitura de temperatura em tempo real.

Projeto 13 – Descrição do código

```
// Termistor_10k Sensor de temperatura
int sensorPin = 1;

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

void loop()
{
  float Valor = analogRead(sensorPin);
  float Volt = 5*Valor/1024;
  float Res=(10000*Volt/(5-Volt));
  Serial.print("Resistencia = ");
  Serial.print(Res,0);
  Serial.print(" ohm\n");
  // Constantes da equação de Steinhart-
Hart
  float a = 0.0011224922;
  float b = 0.0002359132;
  float c = 0.000000074995733;
  float X = b*(log(Res));
  float Y =
c*(log(Res))*(log(Res))*(log(Res));
  float Temp = 1.0/(a + X + Y) - 273;
  Serial.print("Temperatura = ");
  Serial.print(Temp,2);
  Serial.print(" graus C\n");
  Serial.print("\n");
  delay(1000);
}
```

Referências

McROBERTS, M. R. Arduino starter kit manual. A complete beginners guide to the Arduino. Earthshine Design, 2009.

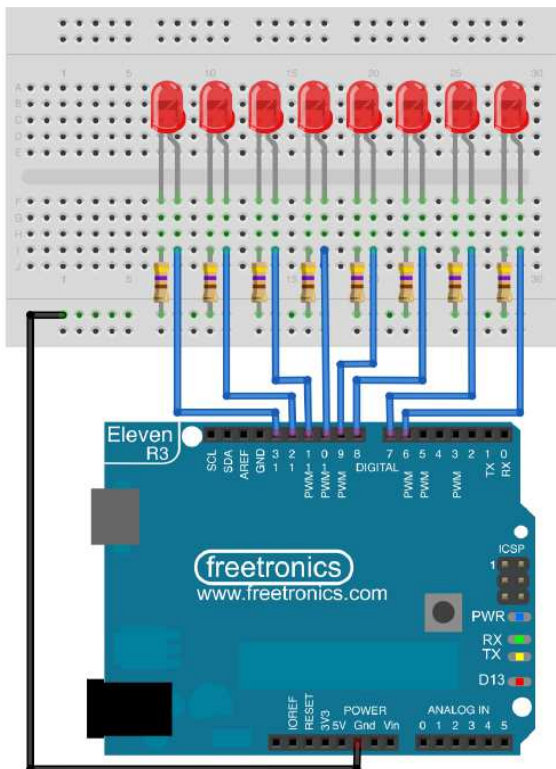
FREETRONICS Experimenters kit project guide v.1.4. Freetronics, Austrália, 2014. Disponível *online* em: <www.freetronics.com/expkit>

ROBOCORE. Arduino kit iniciante v.6.1. Disponível *online* em: <<http://www.robocore.net>>

ELECTRONICS PROJECTS. Learning about electronics. Disponível *online* em: <<http://www.learningaboutelectronics.com/Projects/>>

MICROCONTROLANDO. Utilizando um sensor de temperatura – Termistor NTC 10k no PIC. Disponível *online* em: <<http://microcontrolado.com/termistor-ntc-10k-no-pic/>>

Projeto 15 – Controle de 8 LEDs usando dois laços de repetição



Descrição do código

```
int ledCount = 8;
int ledPins[] = { 6, 7, 8, 9, 10, 11, 12,
13 };
int ledDelay = 300;

void setup() {
  for (int thisLed = 0; thisLed <
ledCount; thisLed++) {
    pinMode(ledPins[thisLed], OUTPUT);
  }
}

void loop() {
  for (int thisLed = 0; thisLed <
ledCount-1; thisLed++) {
    digitalWrite(ledPins[thisLed], HIGH);
    delay(ledDelay);
    digitalWrite(ledPins[thisLed], LOW);
  }
  for (int thisLed = ledCount-1; thisLed
> 0; thisLed--) {
    digitalWrite(ledPins[thisLed], HIGH);
    delay(ledDelay);
    digitalWrite(ledPins[thisLed], LOW);
  }
}
```

This sketch introduces a couple of important concepts in programming: arrays and loops.

The “ledCount” variable specifies how many LEDs are connected, so you can use this same sketch with fewer or more LEDs if you like.

The “ledPins” variable is an “array”, which you can think of as being like a list: in this case it’s a list of the pins that the LEDs are connected to. Later in the sketch we’ll refer to the list to turn different LEDs on and off.

The “ledDelay” variable just sets the number of milliseconds to leave each LED on for. To make the scanner run faster, make the number smaller. To slow it down, make the number bigger.

The setup() function is where things get interesting. It could alternatively have been written with a series of eight nearly identical lines, each one using pinMode() to set one of the LED pins to OUTPUT mode, but that’s not a convenient way to write longer programs. Instead it uses a “for” loop, which runs pinMode() eight times but with a different value each time.

The first argument passed to the “for” loop sets the name of the variable that will be used as the “loop counter”. In this example the loop counter is a number called “thisLed”, and it’s given a starting value of 0.

The second argument sets the terminating condition so the “for” loop will know when it should stop running. The loop will keep repeating itself until the condition fails, so in this case the loop will repeat until the value of “thisLed” is equal to the value of “ledCount”. We’ve set ledCount to 8, so it will keep running until ledPin is equal to 8.

The third and final argument is an action that the loop will perform each time it runs. The “++” operator is a shorthand way of saying to take a variable and add 1 to it, so in this case the “thisLed” variable will get bigger by 1 every time the loop executes.

When you put those three arguments together, you have a loop that will behave in a certain predictable way. It starts out with a variable with a value of 0, increases that value by 1 each time it runs, and when the value reaches 8 it stops. It’s a very concise way of saying “do this particular thing 8 times, slightly differently each time” without having to write each one out individually. The loop itself is interesting, but what’s happening inside it is also interesting. You’ll see it references “ledPins[thisLed]”, which looks a bit confusing.

You’ll remember that the variable “ledPins” is an array, or list, containing a series of values.

Referencing it this way allows us to look at positions in the list without having to know what happens to be on the list in that position. Imagine

you have a restaurant menu with a list of dishes, and each one has a number next to it: you don't need to state the name of the dish, you can just say "I'll have number 4, please" because the position (or number) of the entry is an easy way to reference that item on the menu.

In arrays, the position in the list is called the "array index", and it's a number that specifies what position in the array we want to examine. Unlike typical restaurant menus, though, array indices don't start at 1: they start at 0, so the first position in the array is referenced as index 0. The second position is index 1, and so on. So when the value of "thisLed" is 0, it's the same as referencing "ledPins[0]", which is position 0 (the first item) on the list. The original code looks like this:

```
pinMode(ledPins[thisLed], OUTPUT);
```

The value for thisLed is 0 on the first pass through the loop, so to understand what's going on we can imagine the variable has been replaced by its value, like this:

```
pinMode(ledPins[0], OUTPUT);
```

But of course "ledPins[0]" is really just pointing to the first position in the list. Looking back at the definition of the ledPins array, we can see that the first item on the list has a value of "6". That means the command that is really being executed the first pass is this:

```
pinMode(6, OUTPUT);
```

Then on the next pass through the loop, the value of "thisLed" has increased to 1, so we're referencing "ledPins[1]", which is the second item on the list. Then if you look at the second item on the list and substitute it in, the end result is that what will really be executed on the second pass through the loop is this:

```
pinMode(7, OUTPUT);
```

And so on.

Arrays can be a tricky concept, but they're very powerful and can make your sketches much simpler by allowing you to define lists of things and then step through the list, instead of specifying similar but slightly different operations over and over again and taking many more lines to achieve the same end result.

Next we get into the "loop()" part of the program, which will continue indefinitely. It might look

cryptic at first glance, but when you break it down into its major sections it's fairly simple.

It's really just two more "for" loops, one after the other, that operate just like the "for" loop we looked at a moment ago.

The first loop increments a counter to step forwards through the list using the "++" operator explained previously. It begins at the start of the list of LEDs, and for each one it turns it on, waits for a delay period, then turns it off again before moving on to the next one.

Once the first loop finishes, the second loop begins. The second loop is almost the same, but it runs backwards! It starts at the last position in the list of LEDs, and each time through it uses the "--" operator to decrease the position value by 1. It's like reading the restaurant menu backwards, starting at the last item.

When it reaches the start of the list, the second loop finishes and the program jumps back to the start to begin the first loop all over again.

Projeto 16 – Controle de 8 LEDs usando apenas um laço de repetição

To help understand how the array works, try swapping some of the values in it around. You'll see that the counter will still faithfully step through the list in order, and the LEDs will turn on in the order you specify in the array.

Or you could consider other ways to structure this sketch. For example, think about how you could make it scan from side to side but only use one "for" loop instead of two. At first glance that may seem impossible, but by thinking about how the array is used as a list to step through the LEDs in sequence it's actually quite easy. For example, instead of having a simple list of pins and stepping through it first one way and then the other, you could make a longer array that specifies the complete sequence for an upandback scan, and just loop over it once, like this:

Descrição do código

```
int ledCount = 14;
int ledPins[] = { 6, 7, 8, 9, 10, 11, 12,
13, 12, 11, 10, 9, 8, 7 };
int ledDelay = 300;

void setup() {
for (int thisLed = 0; thisLed < ledCount;
thisLed++) {
pinMode(ledPins[thisLed], OUTPUT);
}
}
```

```

void loop() {
  for (int thisLed = 0; thisLed <
    ledCount1;
    thisLed++) {
    digitalWrite(ledPins[thisLed], HIGH);
    delay(ledDelay);
    digitalWrite(ledPins[thisLed], LOW);
  }
}

```

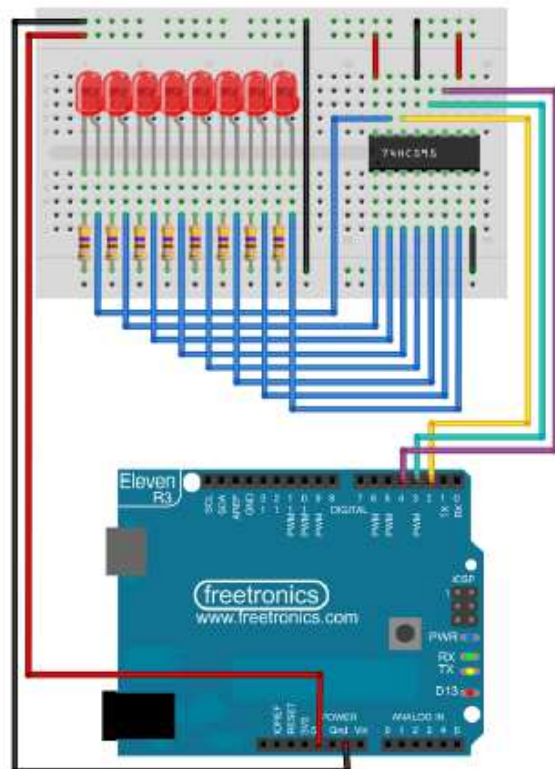
Yes, we've told the sketch that we have 14 LEDs when we really only have 8, but that doesn't matter.

The 14 LED positions count up from 6 to 13 and then back down again, repeating some of the positions. Of course this means that setup() will call pinMode() multiple times on the repeated pins, but that doesn't matter. No harm will be done, and the loop can then step through the list as if it's a linear sequence when in fact the list doubles back on itself.

Projeto 17 – Controle de 8 LEDs usando o CI 74HC595

Microcontrollers have a limited number of input and output pins available for use in your projects, and it's not unusual to run out of pins when you want to control many things at once. A simple solution is a device called a "shift register", which has many outputs that can be controlled individually using just three pins from your Arduino. By sending different values to the shift register you can turn its outputs on or off at will. You can even connect multiple shift registers together in a row called a "daisy chain", giving you the ability to control a huge number of outputs while still using just three pins on the Arduino.

Montagem



Descrição do hardware

If this is your first time using an IC (Integrated Circuit) it may seem a bit scary because it has so many pins, but working with ICs is an important skill to learn.

Before you begin assembly there is one very important thing to remember about most ICs: they can be damaged by invisible static electricity that builds up on your skin, clothes, and workbench. Simply touching the IC in the wrong way can be enough to kill it forever. Some ICs aren't susceptible to static electricity damage, but many are, so the safest thing to do is to assume that all ICs are static-sensitive unless specified otherwise. Don't worry, it's not as bad as it sounds. As long as you follow some simple precautions you'll be fine.

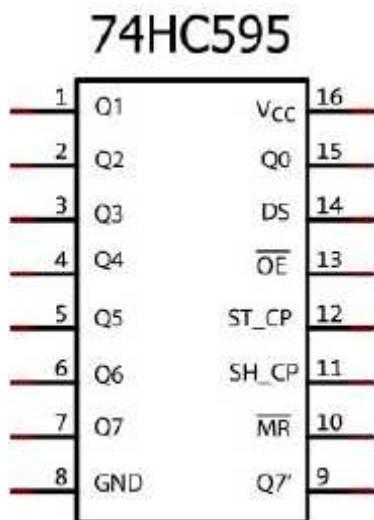
Professional electronics labs often have special work surfaces and antistatic wrist straps, but for experiments on your kitchen table you don't need to go that far. The important thing is to minimize contact with the pins of the IC once it has been removed from its protective foam. Grasp the IC by the ends of the package, and don't touch the pins to anything that may have a static charge. Once you've removed the 74HC595 from its protective foam you'll notice something annoying: the pins aren't straight! IC pins are made to splay out slightly so that they can be inserted into circuit boards with a special tool that squeezes the pins together, and then when the tool

is released the pins spring back out a bit to hold it firmly in place prior to soldering. That means you'll need to bend the pins a bit before you can fit the IC into a solderless breadboard.

To bend the pins, hold the IC package side on against a hard surface such as a wooden table top and push down hard. They'll be quite difficult to bend but you don't want to bend them too far, so the trick is pushing hard enough but not too hard! Try to bend both sides in just enough that they are parallel to each other.

Next, check the orientation of the IC and then push it into the solderless breadboard so that the two rows of pins are on either side of the centre line and the dimple or dot near one end is at the top. This will allow you to easily connect jumper wires to any pin of the IC.

The 74HC595 shift register has a few different features that we're not going to use, so the first thing to do is connect various pins to either GND or 5V to make the chip operate correctly.



With the chip oriented as described above, you'll notice that the bottom left pin connects to GND and the top right pin connects to 5V. This is a loose standard used by most ICs that have two rows of pins. Use jumper wires to connect those pins to the GND and 5V rails on the solderless breadboard.

There are two more pins we need to set to specific values: RESET and OUTPUT ENABLE. Use another jumper wire to connect RESET to 5V, and OUTPUT ENABLE to GND.

Now we're ready to connect up all the LEDs to the outputs of the chip. The outputs are labelled A through H, and each needs to connect to the + (anode) side of an LED. The (cathode) side of each LED then needs to connect to a 470 Ohm resistor (yellowvioletblackblackbrown, or

yellowvioletbrown) which also connects to GND.

The final three connections are the control lines from the Arduino, the vital links that will tell the shift register which outputs should be turned on.

IN (input) to Arduino digital I/O pin D2.

LCLK (latch clock) to Arduino digital I/O pin D3.

CLK (clock) to Arduino digital I/O pin D4.

Once it's all connected up you'll notice there is one remaining pin on the shift register that doesn't have anything connected to it: OUT. That's OK, it's meant to remain unconnected for this project.

Descrição do código

```
int dataPin = 2;
int latchPin = 3;
int clockPin = 4;
void setup() {
  pinMode(dataPin, OUTPUT);
  pinMode(latchPin, OUTPUT);
  pinMode(clockPin, OUTPUT);
}
void loop() {
  for (int currentValue = 0; currentValue <
    256; currentValue++) {
    // Disable the latch while we clock in
    data
    digitalWrite(latchPin, LOW);
    // Send the value as a binary sequence to
    the module
    shiftOut(dataPin, clockPin, MSBFIRST,
      currentValue);
    // Enable the latch again to set the
    output states
    digitalWrite(latchPin, HIGH);
    delay(200);
  }
}
```

Como o software funciona

The explanation behind this sketch may sound a bit brainbending at first, but once you understand what's going on you'll see it's not really that complicated.

The short version is that the 8 outputs of the shift register represent the individual "bit" values of a single "byte" of data. A "byte" is a value that consists in binary form of 8 bits, with each bit having double the place value of the previous bit when read from right to left.

Sound confusing? OK, let's break it down.

One "byte" can be represented in binary form as 8 bits. Each bit can have one of only two values: either 0 or 1, off or on. Just like with regular decimal numbers, the position of each digit changes its value: the number "50" in decimal

means “five tens”, and it’s a bigger number than “5”, which means “five ones”. In just the same way, the binary number “100” (which is not one hundred in decimal, by the way!) has a different value to the binary number “10”, because the place value of the digits is different.

Posição	4	3	2	1
Valor decimal	1000	100	10	1
Valor posição binária	8	4	2	1

Posição	8	7	6	5
Valor decimal	10000000	1000000	100000	10000
Valor posição binária	128	64	32	16

Since a byte has eight bits, and each bit can have a maximum value of 1, the biggest possible number that can fit inside one byte is the binary number “11111111”, which is equivalent to a decimal value of 255: $128 + 64 + 32 + 16 + 8 + 4 + 2 + 1$. One byte can therefore store any number between 0 and 255 decimal.

Now stop for a second and look at that binary number again. That’s eight bits, all in a row, and every bit can have the value of either 0 or 1. Now imagine that each one of those bits is wired up to one of the eight outputs (A through H) of the shift register, so that if that particular bit is set to 1 then the output is on, and if that particular bit is set to 0 then the output is off.

Got it? Well, that’s exactly how a shift register works! The eight outputs are just physical representations of the state of each of the eight bits inside that byte. And by sending a different value for the byte, we can manipulate which outputs are turned on and which are turned off. Remember that the bits are counted from the right in increasing place value (just like decimal numbers) so to set a specific set of outputs on, you just work out the bits to represent those outputs. Want to turn on outputs A, C, and E, and leave all the others off? Easy, just map it out like this, remembering that digits to the left have higher place value so we’ll label it starting with “A” on the right and working to the left like we would with a decimal number:

H	G	F	E	D	C	B	A
0	0	0	1	0	1	0	1

That’s binary “00010101”, and from the previous table we can add up the place values of those bits ($16 + 4 + 1$) to give a decimal value of “21”. So if you send a value of 21 to the shift register, it will turn on outputs A, C, and E, and turn all the others off.

You can use this same technique to figure out how to turn on any combination of outputs, and then map it down to a single value in either decimal or binary that will achieve that end result.

Now with those principles in mind, look at the sketch again and it may make a bit more sense. First it sets up the connections to the shift register input, and to the CLK (clock) and LCLK (latch clock) pins.

Then, in the main loop, it uses a “for” loop to repeat over an incrementing number. The “for” loop starts with a variable called “currentValue” which begins with a value of 0, and keeps repeating as long as the value is less than 256. Each time it repeats, the value increases by 1, so it will go through with a value of 0, then a value of 1, then a value of 2, then a value of 3, and so on until the value reaches 255.

On the next time through the value will reach 256, which hits the limit that has been set and causes the program to move on from the “for” loop. After the “for” loop has finished there are no more commands, so the main loop then just goes back to the start, and begins all over again running the “for” loop from scratch with a starting value of 0 again.

That just repeats forever, so what the sketch does is count from 0 to 255, then go back and do it again.

But the interesting bit is what the sketch does with that value each time it runs through the “for” loop. Inside the “for” loop are three commands: first to change the state of the “LCKL” (latch clock) pin, then a command to send the current value to the register using the IN pin, then to change the state of LCLK back again.

Those three commands, in that sequence, will cause the Arduino to take the value of the “currentValue” variable and send it to the shift register to set the state of all the outputs. Using the “shiftOut” command we can clock a byte of data (value 0 to 255) to the module, setting the state of each output appropriately. For example, sending a value of “0” decimal will set all the outputs LOW. Sending a value of “255” decimal will set all of the outputs HIGH. Sending a value of 1, 2, 4, 8, 16, 32, 64, or 128 decimal will set only output A, B, C, D, E, F, G, or H HIGH respectively.

The sketch below counts from 0 to 255 and sends each consecutive value to the shift register.

Projeto 17 – Usando múltiplos CIs 74HC595

If you want to be able to manipulate individual outputs from within your sketch without affecting the state of other outputs, the most convenient method is probably to store the current value of the shift register as a 1byte variable and apply bitwise operators to it. The Arduino website has a good introduction to bitwise operators at <http://www.arduino.cc/playground/Code/BitMath>.

Sometimes eight outputs aren't enough! You can connect multiple shift registers together in a row (called a "daisy chain") so that you can pass values down the chain to each of the modules individually. That way you can control 16, 24, 32, or even more separate outputs with the same three outputs from your Arduino.

To daisychain multiple shift registers, begin with the simple example above. Then connect a second shift register so that the VCC, GND, RST, OE, CLK, and LCLK connections are all linked to the same place as the first shift register. Finally, run a jumper wire from the OUT connection on the first shift register to the IN connection on the second shift register.

Once you have two shift registers daisychained together, setting their outputs is just as easy as addressing a single register except that you have to send two bytes of data instead of one. The first byte sent will pass down the chain to the last register, and the second byte sent will then go to the first register: think of it as pushing messages down a pipe. The first one you push in will go towards the end, and the second message will go in right behind it.

Using three or more shift registers follows the exact same principle: simply daisychain more together linking the OUT connection of each into the IN connection of the next in the chain, and send as many bytes of data as you have shift registers in the chain.

Addendum:

Understanding Shift Register Connections

Serial OUT: Passes serial data back out to another module. Can be connected to the Serial IN of another module to daisychain them together and control even more outputs. Typically left unconnected.

Serial IN: Data sent from the microcontroller to set the state of the output pins. Connect to a digital I/O pin of your microcontroller.

Latch: Also sometimes referred to as the "storage clock" or "latch clock". On a LOW-to-HIGH

transition this pin causes the values loaded into the shift register to be applied to the outputs. Connect to a digital I/O pin of your microcontroller.

Clock: On a LOW-to-HIGH transition this pin causes the value currently applied to the Serial IN pin to be read into the shift register, and all other values moved along one position.

Output Enable: Active LOW. When pulled HIGH, all outputs are disabled and placed in a high impedance state. When pulled LOW, all outputs are enabled. For minimal pin usage this can be tied to GND to permanently enable the outputs, but this will have the effect that they may start in an unknown state at powerup. Connect to a digital I/O pin of your microcontroller if you want to explicitly enable outputs from software, otherwise connect to GND.

Reset: Active LOW. When pulled LOW, the shift register is cleared and all stored values are lost. Pull HIGH for normal operation. For minimal pin usage this can be permanently tied to 5V to disable reset, but this will require you to explicitly set all 8 bits in order to reset the outputs if they are not in a known state. Connect to a digital I/O pin of your microcontroller if you want to be able to reset the state with a single pulse, but generally it's simplest to connect to 5V.

GND: Connect to GND (0V) on your microcontroller.

VCC: Connect to 5V on your microcontroller.

Outputs A to H: Outputs to your other devices. Can be driven LOW, HIGH, or be set into a high impedance state by setting Output Enable to LOW.