**O arduino**

Para iniciarmos esta jornada são necessárias ferramentas adequadas. A principal, claro, é o **Arduino**. Ao realizar uma pesquisa no Google, você encontrará diferentes tipos de **Arduino** para as mais diversas situações. Como estamos começando, utilizaremos o Arduino Uno, que é a versão de entrada da nossa plaquinha. Na página do Arduinopodemos encontrar outros modelos.



Vamos começar familiarizando-nos com o hardware. Primeiro, precisamos ganhar familiaridade com a parte metálica no canto superior esquerdo da placa. É nela que faremos a conexão USB e ligaremos o **Arduino**. Isso significa que é por ela que iniciaremos a conexão e é nesse ponto que provemos energia para nossa placa.

Um teste simples é pegar um pequeno motor de 5V e ligar o seu fio **preto** em **GND***(terra) e o fio****vermelho****em****5V****. E, caso esteja tudo certo, o motor começará a funcionar! O GND é o famoso*terra\* e ele retira a energia do nosso sistema já o5V, é a porta que coloca energia. Caso você desconecte algum desses fios, o motor parará. Ou seja, precisamos de um circuito fechado para que tudo funcione corretamente!

**Usando protoboard**

Porém, ligar tudo diretamente no **Arduino** não é uma boa prática. Por isso, utilizaremos uma outra peça que nos auxiliará na montagem do nosso circuito, que é a **protoboard**:



**Desenhando cricuitos no Fritzing**

Além da protoboard, precisamos resolver o problema do *desenho dos circuitos*, que ao longo do curso vão se tornando cada vez mais complexos. Para facilitar, utilizaremos uma ferramenta *open-source* chamada **Fritzing**. No site da Fritizing podemos selecionar a opção de download para o SO. No caso do Windows, basta descompactar o .zip que o programa já estará localizado na pasta criada.

A principal vantagem do Fritzing é que ele têm um catálogo de peças eletrônicas bem interessante que utilizaremos no curso. Além disso, ele também é muito intuitivo. Por exemplo, para visualiza a protoboard, basta clicarmos na aba escrito Protoboard.

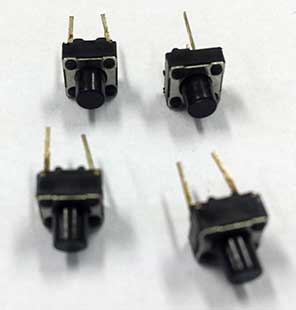
Já foi comentado que conectar diretamente uma peça no **Arduino** é uma má prática e, por isso, precisamos da protoboard. No Fritzing, podemos ligar o nosso motor DC na protoboard e a fileira onde os fios foram conectados poderá receber outras entradas como, por exemplo, o terra e o fio de energia que devem estar ligadas ao Arduino. Para uma melhor realização do processo é de extrema importância a visualização do vídeo para sabermos, corretamente, como dispor as peças na sua protoboard, e como ligar os fios do **Arduino** nela.

Peças obrigatórias para o projeto Genius:

* Arduino Uno R3 com cabo USB (ou outro modelo mais poderoso) 
* Protoboard 
* 4 Resistores de 220 Ohm ou 330 Ohm 
* 4 LEDs comuns 4mm (não precisam ser de auto-brilho), desses:
  + 1 LED vermelho
  + 1 LED azul
  + 1 LED verde
  + 1 LED amarelo



* 4 botões de 2 ou 4 pinos



* Cabinhos de cores variadas (Fio Jumper para protoboard)
  + Azul (2 grandes, 1 pequeno)
  + Vermelho (2 grandes, 1 pequeno)
  + Amarelo (2 grandes, 1 pequeno)
  + Verde (2 grandes, 1 pequeno)
  + Preto (4 grandes, 4 pequenos)
  + Marrom (2 pequenos)
  + Cinza ou Branco (1 grande) 

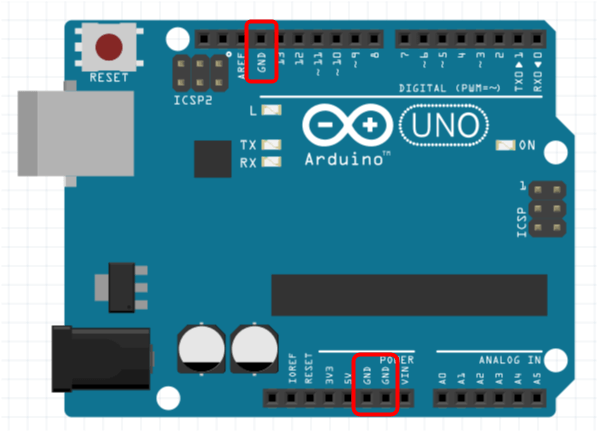
Peças opcionais que usaremos durante o curso mas que não são necessárias para o projeto *Genius*:

* 1 Potenciômetro 10k (opcional)
* 1 Resistor 1kOhm (opcional)
* 1 Motor DC (opcional)
* Multímetro (opcional)

Qual abreviação é usada para a porta Terra na placa Arduino?

GND

A abreviação utilizada é **GND** que significa *ground* ou *terra* em português.

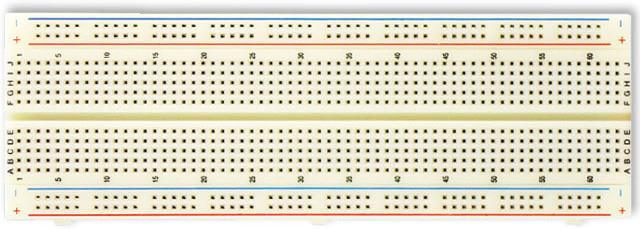


O que é uma **Protoboard**?

A protoboard é uma placa para a criação de circuitos eletrônicos sem soldar os componentes.

A Protoboard é uma placa que possui diversos furos para criar rapidamente circuitos eletrônicos sem ter a necessidade de soldar componentes.

Através dessa placa podemos *prototipar* os circuitos usando cabinhos (também chamados de *fio jumpers*). A Protoboard existe em tamanhos diferentes, para nosso treinamento basta ter um protoboard de tamanho médio, por exemplo:



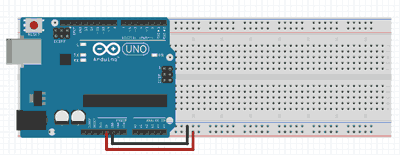
Já vimos que o Arduino também possui alguns conectores (ou portas) de força e terra. Como queremos ligar vários componentes, muitos mais do que existem na placa Arduino. A Protoboard ajuda resolver esses problemas e dar espaço para trabalhar com vários componentes! É como se fosse uma "extensão" com várias tomadas que permitem ligar outros aparelhos.

**Atenção**: Antes de iniciar a montagem do circuito, desligue a conexão USB entre o Arduino e o seu computador!

**Sempre monte todo o circuito com o Arduino desligado. Ligue o Arduino apenas quando terminar a montagem e verificar o circuito.**

Se pesquisarmos por LED no campo de busca acharemos diversos modelos. Escolheremos o mais simples e o ligaremos em nossa protoboard. Porém, como já conectamos o fio GND e 5V no nosso motor dc, como fazemos para ligar os mesmos em nosso LED?

Vamos esticar um novo fio e a partir da coluna ligada do motor o levamos ao LED. Porém, as coisas já estão se complicando e estamos manuseando apenas 2 peças, imagine quando estivermos lidando com várias! Para isso, em vez de ligarmos os fios GND e 5V diretamente na coluna da peça, as ligamos nas **linhas** finais da protoboard. Ligamos, então, o GND na linha mais próxima à linha azul e a saída do 5V conectamos na linha mais próxima à linha vermelha. Seguindo a imagem abaixo:



Perceba que ao ligarmos nessas linhas finais, não é a coluna que fica colorida(ligada), mas sim a linha inteira. Agora, podemos puxar um fio dessas duas linhas da parte de baixo da protoboard e ligá-lo em nossa peça. Como nosso jogo é baseado, justamente, em LED's, precisamos entender como eles funcionam!

## Sobre o LED

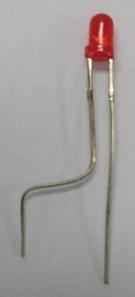
A sigla LED significa Light Emitting Diode (Diodo Emissor de Luz). Um LED é usado para emissão de luz em locais e também em instrumentos onde se torna mais conveniente sua utilização do que a de uma lâmpada. E Diode, você sabe o que é? Diode é um componente eletrônico que faz com que a corrente elétrica circule em sentido único, nunca permitindo que ela circule na direção oposta.

Certo, mas como visualizar isso em um LED? Vejamos a seguinte foto:



Perceba que o LED possui uma perna maior do que a outra. Por quê? É uma maneira de distinguir seu polo positivo do negativo. A perna maior sempre será positiva ( nodo) e a menor sempre negativa (Cátodo). É por esse motivo que dizemos que o LED é polarizado, pois, ele possui dois polos, um positivo e outro negativo.

Já tentou colocar um LED em pé? Por possuir uma perna maior que a outra, essa será uma tarefa difícil de ser executada, por isso é comum entornarmos a perna maior para que ambas fiquem com o mesmo tamanho o que facilita seu encaixe na protoboard. Vejamos um LED que teve sua perna maior dobrada:



Ao ligar o LED na protoboard e conectar os fios, passaremos por outro problema. A Amperagem do LED(vermelho) é de **15 mA\***(quinze mili-ampére). E o nosso ***Arduino***emite ***40mA***, ou seja, corrente suficiente para queimar o nosso LED. Para diminuirmos a quantidade de corrente passando pelo LED, utilizaremos outra peça famosa, o ***resistor***. Para entender o resistor, precisamos lembrar da Lei de Ohm, lááááááá do ensino médio. Essa lei faz a relação entre **tensão**, **resistência** e **intensidade da corrente**, respectivamente. Observe sua fórmula:

V = R \* I

Em nosso circuito, temos a saída do **Arduino** com 5V. Porém, precisamos contabilizar que ao ligar o LED vermelho, ele também consome um pouco dessa voltagem. Precisamente, o LED vermelho absorve 2V. Substituindo na nossa fórmula, temos:

5-2 = R \* I

3 = R \* I

Agora, precisamos olhar para a corrente elétrica(**I**). Conforme mencionado anteriormente, o **Arduino** nos fornece uma corrente elétrica de **40 mA**. Porém, não utilizaremos esse valor na fórmula e sim o da **corrente final**. No caso, a corrente que o LED vermelho necessita é de **15mA**. Substituindo novamente na nossa fórmula, e realizando a conta, teremos o seguinte:

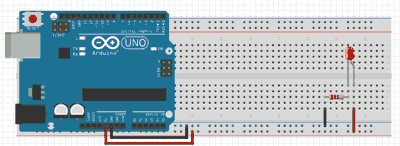
3 = R \* 15mA

3 = R \* 0,015A

R = 3 / 0,015

R = 200 ohm

Vimos que a resistência necessária para que os LEDs não queimem é de no mínimo **200 ohm**. Entretanto, procurando na lista de resistências do Fritzing não encontramos o valor de 200. Como estamos falando de resistência o melhor é errarmos para cima***, ou seja, podemos selecionar a primeira resistência após o valor de 200. No Fritzing, essa resistência é a de***220 ohm\*. Após isso, basta ligarmos a nossa resistência, como na imagem:



A tensão elétrica (V) é medida em *Volts (V)* (em homenagem ao físico *Alessandro Volta*).

A resistência (R) é medida em *Ohm (Ω)* e a intensidade da corrente elétrica (I) em *Ampere (A)*.

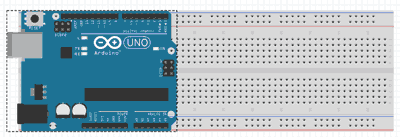
A relação entre **tensão**, **resistência** e **intensidade da corrente** define a *Lei de Ohm*: **V = R \* I**

1) Instale o programa **Fritzing** em seu computador. Basta baixar o link abaixo e descompactar:

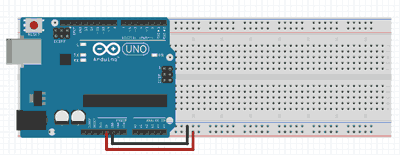
<http://fritzing.org/download/>

2) Abra o Fritzing e crie um novo projeto (Arquivo -> Novo)

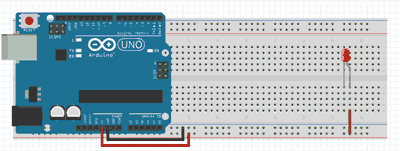
3) Na aba *Protoboard*, no lado direito, procure pelo **Arduino UNO** e arraste a protoboard:



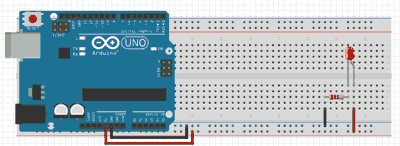
4) Alimente as linhas inferiores da protoboard com a força (5V) e terra (GND):



5) Procure pelo LED no Fritzing e arraste o LED vermelho para a protoboard. Alimente o LED com a força na perna comprida.



6) Procure pelo resistor no Fritzing e arraste também na protoboard. Coloque o resistor na perna curta do LED e ligue na terra:



Seguem algumas regras para auxiliar a montagem do circuito real:

* Sempre desenhe o circuito, usando Fritzing, antes de iniciar a montagem!
* A energia deve circular da força para terra.
* Use um resistor de 220Ohm ou 330Ohm para proteger o LED
* O resistor pode aparecer antes ou depois do LED, não faz diferença
* A perna comprida do LED é a *perna positiva*
* Sempre use cabos com cores, isso ajudará a compreender o circuito
* Lembre-se da separação da protoboard em linha e colunas
* Já usamos as portas de 5V e GND do Arduino

## Vimos ao longo dessa aula a como identificar quantos *Ohm* um resistor possui. Mas, é sempre bom revisar o aprendizado!

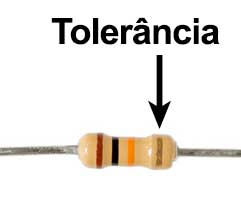
## Na aula o instrutor apresentou uma tabela de cores que serve como base para o cálculo:

## https://lh4.googleusercontent.com/CUrWb93ntYeXjm_Z4a2lNuDntBF50f7bKiOnCfgvl7oYnDvWDOd4JRuEWDY_N_Jd4Kdl-o36z20ucxWOPWXxrgpimrWzjnY9WJneB8FTAxRK4XblukLzhlEs5_9uM9noL-2PhH5o

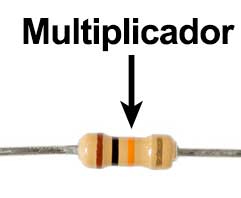
## Conhecendo essa tabela, *quantos* Ohm *o resistor* abaixo possui?

## https://lh6.googleusercontent.com/OJUTYO_ghZEfSXkzJCNKMzUQPsoqbr3Hlxao_CDmuKwix9CwxXDL7GWPlLOQAdJatLuOph8HwkI-RhnhBgbGBhlQ1q6Gh6N_sIjMvBA19Xq7OHf2UVK7AnOWF1ZOXAmn2fOj0sS4

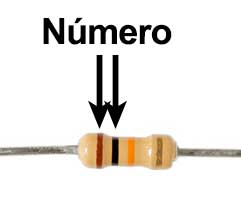
No resistor apresentado na imagem temos uma cor que se encontra no lado direito e que representa a tolerância. Aqui, a **cor ouro** mostra o valor final que pode variar de 5% para cima ou para baixo. Se a cor fosse prata haveria uma variação de 10%.



Agora, observe a próxima faixa ao lado esquerdo da cor ouro, é nela que descobriremos o multiplicador. A cor dessa faixa é **laranja** e procurando o valor associado a ela na tabela descobrimos que seu multiplicador é de 1kΩ.



Sobram mais duas faixas que representam o outro coeficiente da multiplicação. A primeira faixa da esquerda tem a **cor marrom**, a segunda é **preta**. Nesse exemplo temos apenas duas faixas sobrando e essas duas juntas criam um número. A cor marrom representa *1* e a cor preta *0*, então, o número é **10**.

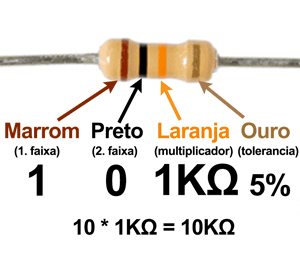


Analisadas essas questões temos tudo para calcular. O número é 10, o multiplicador é 1kΩ, portanto:

10 \* 1kΩ = 10kΩ

**O resultado é 10kΩ com 5% de tolerância.**

Veja também a imagem abaixo para comparação:



A conta pode parecer complicada no início, mas na verdade não o é! O maior problema é enxergar as cores nessas faixas do resistor :) Uma vez identificado o que é faixa e o que é multiplicador, os mistérios se diluem e a conta se torna mais fácil!

Vamos revisar rapidamente os pontos principais do Lei de Ohm:

Por padrão, os elétrons se movem desordenadamente em todas as direções, porém, quando conseguimos fazer com que eles sigam uma direção, conseguimos gerar uma corrente elétrica.

A **corrente elétrica** é o movimento de elétrons por um condutor, por exemplo, um fio de cobre. Sua medição é feita a partir da quantidade de elétrons que passam por um ponto em determinado momento, isto é, sua intensidade. Quanto mais elétrons (carga) passarem em um ponto, mais intensa será corrente. A unidade de medida da corrente é o ampère (A). Mas como conseguimos fazer com que todos os elétrons sigam uma direção? É aí que entra em ação a **tensão**.

O resultado da diferencial de potencial (DPP) entre dois pontos é o que resulta na tensão. Sua unidade de medida é **volts**. Um exemplo desses "dois pontos" é uma pilha. A falta de elétrons em um pólo e o excesso em outro origina uma diferença de potencial (DDP). É por isso que quando ligamos um motorzinho a uma pilha, a corrente do pólo positivo passará pelo motor até chegar ao polo negativo da pilha. Se a pilha não tivesse seu pólo negativo, não haveria corrente, sem corrente, o motor não funcionaria.

No caminho percorrido pela corrente pode haver uma resistência, algum componente que não gosta de corrente elétrica e que fica "quente de raiva". Essa resistência diminui a corrente que passa por ela e, devido a essa resistência, a amperagem acaba ficando menor. Usamos o termo amperagem porque a corrente é medida em ampères (A). Mas por que precisamos saber disso tudo? **Para não torrarmos nossos projeto!**

Vamos voltar ao problema que envolve a ligação de um LED em nossa protoboard. A porta analógica do Arduino possui uma **tensão** de 5 volts e corrente de 40 miliampère (mA). Nosso LED vermelho possui tensão de 2 volts e suporta uma corrente de 20 (mA). Como sabemos desses dados de tensão e amperagem? Consultando a documentação do Arduino e a especificação do LED que estamo usando.

Como o Arduino tem amperagem de 40mA, ela será suficiente para danificar nosso LED, que só sustenta 20mA. Precisamos de alguma maneira reduzir a corrente (ou amperagem, se você preferir) que chega até o LED. Mas qual resistência colocar, uma de 20mA? Infelizmente a lógica não é assim. A resistência tem outra unidade de medida, chamada **ohm**. Seu símbolo é a ferradura Ω. Precisamos chegar ao valor dessa resistência através de uma fórmula, a famosa **Lei de Ohm**.

R = V/I

* **V** = tensão em volts (ou voltagem, no popular)
* **R** = Resistência em Ω
* **I** = intensidade da corrente em ampères

E se eu quisesse calcular não a resistência, mas a corrente? E se eu quisesse calcular a tensão? Dependendo do que queremos calcular a mesma equação pode ser de três formas:

**para calcularmos a corrente**

I = V / R

**para calcularmos a resistência**

R = V / I

**para calcularmos a tensão**

V = R \* I

Se estamos ligando a porta de 5v com o LED de 2V, precisamos entender que o LED consumirá 2 volts da corrente de 5V. Então, em nossa fórmula temos:

R = (5 -2) / I

Excelente, sabemos que a corrente máxima que pode chegar até nosso LED é de 20mA. Sendo assim, podemos colocar os 20mA em nossa fórmula não como miliampère, mas como ampère:

R = (5 -2) / 0,02

Agora, só resolver a equação:

R = 3 / 0,02  
R = 150

E se não tivermos uma resistência de 150Ω? Não se preocupe, você pode usar uma resistência mais alta, mas nunca uma mais baixa. No máximo isso influenciará na intensidade da cor do LED. Em nossa caso, temos resistores de 220Ω que resolvem a problemática dos LEDs.

## Já ouviu falar de Raspberry PI? Raspberry PI é uma outra placa, parecida com o Arduinoe que também serve para criação de diversos projetos!

## Em geral, o Arduino e o Raspberry PI fazem parte do mundo dos entusiastas da tecnologia e do *DIY* (*Do It Yourself* - faça você mesmo). Ambos permitem a execução de um programa e a conexão com componentes através de portas para controlar o "mundo lá fora" . Contudo, há uma diferença entre essas plataformas! Qual é?

## Se estiver com dúvida pesquise sobre o tópico! Ou, clique em *continuar* para ver a resposta ;)

## https://lh4.googleusercontent.com/83T6CpIlGiSYfWZhZjZgVvAyGj19gTbUhY19Gq-EaQdKn59mTWrAXZotzOkSVRXG1QN1RVYKRBNaroOCUs-JAuWX3WMu327DoA75Wa4QeeNwrmzIZ2f5zXNbRU9pOpxTmfBCkTx6

## A resposta simplista a essa pergunta é:

## Arduino é um microcontrolador e Raspberry PI é um single board computer.

## Mas, afinal, o que isso significa?

## Usando o Raspberry PI você roda, na verdade, um sistema operacional normalmente, o Linux. Dessa maneira você terá uma interface gráfica, poderá instalar outros softwares, será permitido conectar mouse ou teclado e assim por diante. Ou seja, você terá acesso a tudo aquilo com o que já está acostumado em seu próprio computador!

## Mas, no Arduino não há um sistema operacional, nem conectores USB, muito menos saída HDMI ou rede. Para executar um programa, você deverá escrever o seu código que será compilado (traduzido) para a linguagem do microcontrolador e este código será rodado diretamente no próprio Arduino. Veremos mais sobre isso ao longo do curso!

## Enquanto o Arduino foi criado para controlar dispositivos através das portas, o Raspberry PI tem uma abrangência muito maior, pois é um mini computador. Então, o mais correto seria comparar o Raspberry PI com um computador, uma vez que o Arduino foi criado especificamente para projetos eletrônicos.

## Por isso o Arduino tem muito menos RAM (somente 2k), apenas 16MHz usando 8bit, enquanto o novo modelo do Raspberry PI tem um quadro-core 1.2GHz, 1 GB de RAM e 32bit. Se o seu foco for criar projetos eletrônicos, o Arduino é normalmente suficiente, mas se você também deseja rodar um servidor web ou trabalhar com câmera, o mais recomendável é utilizar o Raspberry PI.

## Isso também explica a diferença de preços entre as placas. No momento da elaboração desse texto o Raspberry PI custava 8 vezes mais caro do que o Arduino.

## Então, o Arduino é muito mais barato, pois é mais simples, mas será que ele tem outras vantagens?

## Claro! Ele foi pensado para *controlar* componentes eletrônicos como, sensores (principalmente análogos), motores, LEDs, botões, etc. Por isso, ele possui uma tolerância e flexibilidade maior. Por exemplo, por padrão, o Arduino é capaz de receber valores analógicos de um sensor, enquanto Raspberry PI só trabalha com uma entrada digital.

## Isso significa, ao mesmo tempo, que você precisa ter algum conhecimento sobre eletrônica para manusear o Arduino, enquanto no Raspberry PI, isso não é necessário. Nessa área existem outras diversas diferenças, mas um dos pontos positivos do Arduino é ser mais flexível e robusto, quando falamos da parte de eletrônica.

## Finalizando, se você deseja trabalhar apenas com eletrônica (sensores, motores, etc), o Arduino pode ser mais adequado para suas demandas. Agora, se você estiver interessado em usar software como servidor web, integrado com câmera e rede, usar um media center ou rodar um software mais complexo, o Raspberry vai te atender bem.

## Para finalizar mesmo... Também existem projetos que usam as duas placas juntas :)

## Vamos continuar construindo nosso Genius. Repare no circuito que desenvolvemos desde o capítulo 1, nele conectamos o LED na porta de 5V. Porém, nós não temos *controle* sobre essa porta o que significa que não temos como alterá-la, assim ela sempre irá fornecer 5V. Para termos esse controle de falar para a porta o que ela deve fazer, precisamos usar as *portas digitais*. As portas digitais ficam do lado contrário a de 5V.

## Atenção: as portas 0 e 1 não devem ser usadas neste momento, por isso, comece pela porta 2.

## Agora que nosso fio está conectado a porta 2 e não mais na de 5V, precisamos criar os comandos de ação para essa porta. Para isso, precisaremos *programar* esse comportamento. A linguagem de programação usada no Arduino é a C++, nós utilizaremos a IDE oficial, que pode ser baixada [aqui](https://www.arduino.cc/en/Main/Software).

## Antes de começarmos a programar o ideal é que você observe [esta página](https://www.arduino.cc/en/Reference/HomePage). Nela estão armazenadas todas as documentações de funções, constantes, variáveis, estruturas e muito mais. Nesse curso tentaremos cobrir tudo o que utilizarmos porém, essa é uma dica valiosa para seus projetos futuros.

## Começando a Programar

## Ao abrir a IDE do Arduino nos deparamos com duas funções: setup() e loop(). O código dentro de setup() será executado uma vez, já o código contido no loop(), como o próprio nome insinua, será executado infinitas vezes até pararmos ele.

## Queremos comunicar ao Arduino que "Nós temos uma conexão na porta 2!". Consultando a documentação encontramos a função pinMode(), que recebe a porta e indica se ela é de saída ou entrada. Vamos falar, então, que nossa porta 2 é de saída, assim, a partir dela emitiremos o sinal para o LED acender.

## void setup(){    pinMode(2, OUTPUT); }

## Após a codificação, nós compilamos o código para verificar se tudo está em ordem com a sintaxe. Feito isso é necessário verificar se estamos lidando com a placa correta. Vamos em Ferramentas -> Placa ... e averiguamos se nosso modelo está selecionado. Depois, logo abaixo, verificamos se o SO achou a porta certa do Arduino. Ele deve fazer isso automaticamente.

## Feito isso, podemos voltar a programar! No caso, queremos que a luz acenda! Dessa forma, é preciso escrever um sinal para a nossa porta. Usaremos a função digitalWrite(), que recebe o sinal e também a porta para a qual enviaremos o sinal. Como desejamos que acenda, enviaremos o sinal 1.

## void setup(){    pinMode(2, OUTPUT);    digitalWrite(2,1); }

## Agora, já podemos verificar se funciona! Ao compilarmos novamente e ao clicarmos na seta ao lado, que envia o programa para o Arduino, veremos a luz acender! Porém, dessa vez, fomos nós que controlamos a saída de sinal. Por exemplo, se quiséssemos acender e apagar o LED em seguida, bastaria repetir a linha e mandar um sinal 0.

## void setup(){    pinMode(2, OUTPUT);    digitalWrite(2,1);    digitalWrite(2,0); }

## Compilando e mandando esse código para o Arduino apenas aparecerá que a luz ficou apagada. Isso acontece porque logo depois que ela acende, ela também apaga! Só que isso acontece muito rápido! Para resolvermos essa questão podemos inserir um intervalo de tempo entre as duas ações:

## void setup(){    pinMode(2, OUTPUT);    digitalWrite(2,1);    delay(1000);    digitalWrite(2,0); }

## A função delay() recebe como parâmetro o tempo, que deve ser escrito em *milisegundos*. Logo, ao escrever delay(1000), isso significa que o tempo de espera será de *1 segundo* até que a luz se apague. Mas, note que quando usamos sinal 1 ou sinal 0não estamos aplicando códigos muito intuitivos ou legíveis. Portanto, podemos substituir esses nomes por *constantes* do Arduino, as chamadas HIGH e LOW que significam, respectivamente, *ligado* e *desligado*. Observe como fica substituindo isso em nosso código:

## void setup(){    pinMode(2, OUTPUT);    digitalWrite(2,HIGH);    delay(1000);    digitalWrite(2,LOW); }

## Repare que repetimos o valor 2, que é a nossa porta digital. Será que outra pessoa ao ler nosso código, entenderia isso de primeira? Acho que não! Por isso, podemos acoplar esse valor 2 em uma variável. Como nosso código está escrito em linguagem C++, é preciso acrescentar um tipo. Esse tipo será *inteiro*, no caso, int. Assim, em todo lugar que quisermos representar nossa porta deveremos adicionar o nome dessa variável. Ficará assim:

## void setup(){    int ledVermelho = 2;    pinMode(ledVermelho, OUTPUT);    digitalWrite(ledVermelho,HIGH);    delay(1000);    digitalWrite(ledVermelho,LOW); }

## Passando para o loop

## Caso surjam dúvidas ao longo do processo basta compilar e testar para verificar se continua funcionando!

## Agora, vamos reparar outro *erro* nosso! Queremos que a luz pisque. Porém, o código do setup() roda apenas uma vez. Logo, fica fácil resolver isso, basta passar a utilizar o código de acender e apagar a luz junto com o loop(). Porém, ao fazer isso, a função loop() não irá enxergar a variável ledVermelho, que ficou no setup(). Para solucionar esse problema faremos com que a variável seja global, o que significa que ela será "vista" por qualquer função. Além disso, vamos adicionar um tempo para a *luz apagada*também. Vamos então ao código:

## //variável global int ledVermelho = 2; void setup(){    pinMode(ledVermelho, OUTPUT); } void loop(){    digitalWrite(ledVermelho,HIGH);    delay(1000);    digitalWrite(ledVermelho,LOW);    delay(500); //espera meio segundo antes de acender novamente }

## Para começar a desenvolver a parte de programação do nosso projeto, precisamos utilizar um software que ajude a criar o código específico para Arduino. Esse software chama-se IDE: *Integrated Development Environment* ou *Ambiente de Desenvolvimento Integrado*.

## Neste curso, utilizaremos a IDE oficial do Arduino, que pode ser obtida gratuitamente do [Site oficial do Arduino](https://www.arduino.cc/en/Main/Software).

## Lá você pode escolher entre as versões de Windows, Mac ou Linux , de acordo com o sistema operacional de sua preferência. Caso você use o Ubuntu, e ele apresente o seguinte erro ao detectar a sua porta USB:

## https://lh5.googleusercontent.com/yKNc0K4N_atulxUJCwdvR3PyCyNMVs1zuPF3NC2O9Ii90f1NpZQFSIcnJsyp_xdJei6YVxcEF8EB9WVS-zw6N6I3LOGPidCgnXDXQqzJByZOKRZUWyPZ21ppjrCaPHGu-pKs1aHx

## Abra o terminal e execute o seguinte comando, substituindo SEU\_USUARIO pelo nome do seu usuário:

## sudo usermod -a -G dialout SEU\_USUARIO

## Isso deve solucionar o problema! Após fazer isso, encerre a sessão do seu usuário e faça o login novamente.Agora, você já poderá selecionar a sua porta USB na IDE do Arduino.

## Diversas vezes não sabemos o porquê do código não funcionar. Para nos auxiliar nessa tarefa existe o Monitor Serial. Nele, podemos printar, ou seja, escrever o valor de variáveis e outras mensagens que nos guiarão pelo programa.

## Para utilizá-lo é preciso, primeiro, iniciá-lo com a função Serial.begin(). Essa função recebe como parâmetro a *taxa de transferência* que pode ser verificada no próprio monitor serial, no canto inferior esquerdo. Em nosso caso, essa taxa é de 9600. Para escrever algo dentro do monitor basta usar a função Serial.println() e, por exemplo, se quisermos saber o valor da variável ledVermelho a cada iteração do nosso loop(), podemos usar o seguinte código:

## //variável global int ledVermelho = 2; void setup(){    Serial.begin(9600);    pinMode(ledVermelho, OUTPUT); } void loop(){    Serial.println(ledVermelho);    digitalWrite(ledVermelho,HIGH);    delay(1000);    digitalWrite(ledVermelho,LOW);    delay(500);    ledVermelho++; //aumenta o valor de ledVermelho em 1 }

## Dentro do Monitor Serial podemos verificar o valor de ledVermelho aumentando. Além disso, a luz para de piscar após a primeira interação. Isso acontece pois o valor de ledVermelho muda, conforme podemos verificar, logo, a porta indicada para ligar e desligar é a porta errada.

## Utilizando constantes

## Com esse pequeno experimento percebemos que, na verdade, o valor de ledVermelhonão deveria ser uma variável, mas sim uma constante. Isso significa que precisamos fazer com que o valor dela nunca mude. Podemos, ainda, extrair outras constantes como o segundo e o meio-segundo. O jeito de *definir* isso no C++ é bem simples. Por convenção, toda constante fica completamente em caixa-alta. Portanto:

## //int ledVermelho = 2; //declarando a constante #define LED\_VERMELHO 2 #define UM\_SEGUNDO 1000 #define MEIO\_SEGUNDO 500 void setup(){    Serial.begin(9600);    pinMode(LED\_VERMELHO, OUTPUT); } void loop(){    Serial.println(LED\_VERMELHO);    digitalWrite(LED\_VERMELHO,HIGH);    delay(UM\_SEGUNDO);    digitalWrite(LED\_VERMELHO,LOW);    delay(MEIO\_SEGUNDO); }

## Nos preparando para os próximos LEDs

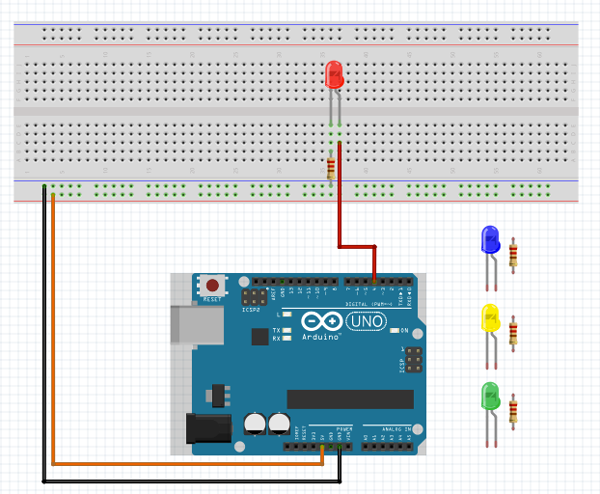
## Nesse momento o código está funcionando como queremos. Porém, nosso jogo pode ter *(e terá)* diversos LEDs. Se formos copiar e colar cada linha de código para cada LED diferente a chance de cometer algum erro é gigante. Por isso, criaremos funções que irão encapsular, ou seja, guardar um código específico para nós. Assim, quando elas forem chamadas executarão o código normalmente. Vamos fazer isso com o código de piscar o LED?

## //int ledVermelho = 2; //declarando a constante #define LED\_VERMELHO 2 #define UM\_SEGUNDO 1000 #define MEIO\_SEGUNDO 500 void setup(){    Serial.begin(9600);    pinMode(LED\_VERMELHO, OUTPUT); } void loop(){    Serial.println(LED\_VERMELHO);    piscaLed(); //estamos chamando a função } void piscaLed(){    digitalWrite(LED\_VERMELHO,HIGH);    delay(UM\_SEGUNDO);    digitalWrite(LED\_VERMELHO,LOW);    delay(MEIO\_SEGUNDO);     }

## Perceba que já usamos diversas funções em nosso código, por exemplo, o pinMode(). Dentro dos parênteses passamos alguns valores como LED\_VERMELHO e OUTPUT. Podemos mudar a função para que pisque a cor específica do led que indicamos. Para isso, vamos passar parâmetros no loop() e declará-los na nossa função mesmo. Fica assim:

## //int ledVermelho = 2; //declarando a constante #define LED\_VERMELHO 2 #define UM\_SEGUNDO 1000 #define MEIO\_SEGUNDO 500 void setup(){    Serial.begin(9600);    pinMode(LED\_VERMELHO, OUTPUT); } void loop(){    Serial.println(LED\_VERMELHO);    piscaLed(LED\_VERMELHO); //estamos chamando a função } void piscaLed(int portaLed){    digitalWrite(portaLed, HIGH);    delay(UM\_SEGUNDO);    digitalWrite(portaLed, LOW);    delay(MEIO\_SEGUNDO);     }

No capítulo passado ajeitamos o LED Vermelho e o programamos para que ficasse piscando. No caso, nosso circuito ficou da seguinte maneira:



## Inserindo outros LEDs

Agora vamos organizar nosso circuito*(no Fritzing!)* para que possamos colocar os **4 LEDs**no jogo. Para tanto, utilizaremos a mesma ideia do primeiro, sempre deixando um espaço igual entre cada um dos LEDs, para que o circuito não fique bagunçado. Colocaremos na ordem: *verde, amarelo, vermelho, azul*. Teremos:



Perceba as **conexões**:

* LED Verde na porta 2;
* LED Amarelo na porta 3;
* LED Vermelho na porta 4;
* LED Azul na porta 5;

É importante que o circuito esteja nessa mesma ordem, pois isso interfere na programação do **Arduino**!

Após a adição desses outros LEDs podemos testar o código e ver como ele funciona. Perceba que na verdade quem acende é o LED Verde, pois é ele que está mapeado na porta 2. Os outros LEDs não realizam ação, uma vez que não foram programados para isso.

## Programando os Novos LEDs

Vamos partir para a ação? Devemos indicar ao nosso programa que temos 4 novos LEDs. Portanto, é preciso definir mais **3** constantes no nosso programa, além disso, não se esqueça de ajustar o número do LED Vermelho.

#define LED\_VERDE 2  
#define LED\_AMARELO 3  
#define LED\_VERMELHO 4  
#define LED\_AZUL 5

Agora, repetimos o processo que fizemos com o LED Vermelho. Ou seja, precisamos falar para o **Arduino** que os nossos LEDs são uma saída e não entrada de dados. Por isso, dentro da nossa função de setup() teremos o seguinte:

void setup(){  
    Serial.begin(9600);  
    pinMode(LED\_VERDE, OUTPUT);  
    pinMode(LED\_AMARELO, OUTPUT);  
    pinMode(LED\_VERMELHO, OUTPUT);  
    pinMode(LED\_AZUL, OUTPUT);  
}

Depois de inserir o setup podemos utilizar a função piscaLed() que indicará para cada LED que ele deve piscar, em loop(). Vamos adicionar também um delay() no final, assim, poderemos ver claramente como o código está acontecendo. Ficará assim:

void loop(){  
    piscaLed(LED\_VERDE);  
    piscaLed(LED\_AMARELO);  
    piscaLed(LED\_VERMELHO);  
    piscaLed(LED\_AZUL);  
    delay(MEIO\_SEGUNDO);  
}

Observe que devido as nossas definições o código ficou bastante expressivo. Até mesmo uma pessoa que não saiba programar, consegue compreender que ele pisca cada LED em sequência e depois dá uma pausa de MEIO\_SEGUNDO.

E se quiséssemos uma sequência diferente? Teríamos que mudar a ordem de chamada da função piscaLed(), não é mesmo?! Antes, vamos tentar uma abordagem um pouco diferente! Vamos começar criando uma nova função: piscaSequencia1(). Dentro dela nós só copiaremos o conteúdo que tínhamos no loop().

void loop(){  
    piscaSequencia1();  
}  
  
  
void piscaSequencia1(){  
    piscaLed(LED\_VERDE);  
    piscaLed(LED\_AMARELO);  
    piscaLed(LED\_VERMELHO);  
    piscaLed(LED\_AZUL);  
    delay(MEIO\_SEGUNDO);      
}

Já conseguimos desacoplar, ou seja, modularizar nosso código, uma vez que é preciso apenas chamar a nova função para que todas as luzes pisquem numa sequência. Mas e se quiséssemos acender todas as luzes ao mesmo tempo? Para isso, vamos dar uma arrumada na casa e criar uma nova função, que realize essa tarefa para nós:

void loop(){  
    piscaSequencia1();  
    piscaSequencia2();  
}  
  
  
void piscaSequencia2(){  
    digitalWrite(LED\_VERDE, HIGH);  
    digitalWrite(LED\_AMARELO, HIGH);  
    digitalWrite(LED\_VERMELHO, HIGH);  
    digitalWrite(LED\_AZUL, HIGH);  
  
    delay(UM\_SEGUNDO);  
  
  
    digitalWrite(LED\_VERDE, LOW);  
    digitalWrite(LED\_AMARELO, LOW);  
    digitalWrite(LED\_VERMELHO, LOW);  
    digitalWrite(LED\_AZUL, LOW);  
  
    delay(MEIO\_SEGUNDO);  
}

Nessa nova função todas as luzes se acendem e, então, aguardamos por 1 segundo e apagamos todas elas. Teste no seu **Arduino**!

## Se aproximando mais do jogo

Agora temos um conhecimento maior sobre como manipular os LEDs, tanto na parte de *eletrônica* quanto na parte da *programação*. Uma lição importante é que sempre podemos melhorar um pouco o nosso código! Por exemplo, não faz parte do jogo as luzes piscarem assim que o **Arduino** é ligado. Além disso, na setup() existe muita configuração de porta. Por que não tentamos arrumar isso?

void setup(){  
    Serial.begin(9600);  
    iniciaPortas();  
}  
  
  
void iniciaPortas(){  
    pinMode(LED\_VERDE, OUTPUT);  
    pinMode(LED\_AMARELO, OUTPUT);  
    pinMode(LED\_VERMELHO, OUTPUT);  
    pinMode(LED\_AZUL, OUTPUT);  
}  
  
  
void loop(){  
    //retirado o código de testes anteriores  
}

## Nosso array de LEDs

A função iniciaPortas() faz a configuração inicial para nós. Para que fiquemos mais próximos do jogo, precisamos perceber a nossa necessidade, ou seja, **gerar** uma sequência de luzes a qual o jogador terá que imitar. Para fazer tal feito em *C++* é preciso criar um array ou vetor, ou seja, uma variável que guarde o valor de **diversos** LEDs e não apenas de 1.

//nosso array de LEDs  
int sequenciaLuzes[] = [LED\_AZUL, LED\_VERDE, LED\_VERMELHO, LED\_AMARELO];  
  
  
void loop(){  
    piscaLed(sequenciaLuzes[0]);  
}

Perceba que após a variável foi adicionado o sequenciaLuzes, dois colchetes e um número dentro. Esta é a forma que indicamos exatamente **qual** LED queremos que pisque. A contagem sempre começa pelo **0**, então, no caso, mandamos piscar o LED\_AZUL. Caso quiséssemos o LED\_VERMELHO, indicaríamos entre os colchetes o número **2**.

Mas, e se quiséssemos mudar a sequência? E se quiséssemos que a sequência fosse iniciada somente quando o jogador clicasse em algum botão? Para fazer isso é preciso isolar essa parte da programação. E como já aprendemos, *isolar = criar uma função*. Observe:

//definimos o tamanho da nossa sequência  
#define TAMANHO\_SEQUENCIA 4  
  
  
int sequenciaLuzes[TAMANHO\_SEQUENCIA];  
  
  
void iniciaJogo(){  
    sequenciaLuzes[0] = LED\_AZUL;  
    sequenciaLuzes[1] = LED\_VERDE;  
    sequenciaLuzes[2] = LED\_VERMELHO;  
    sequenciaLuzes[3] = LED\_AMARELO;  
}

Veja que definimos também o tamanho da nossa sequência. Fizemos isso, pois no *C++*, não podemos declarar um array sem definir o seu tamanho. A única exceção, inicializarmos a variável logo no seu momento de criação, que nem estávamos fazendo antes. Porém, como queremos mudar seu conteúdo, e inicializar com valores diferentes mais pra frente no jogo, precisaremos indicar o seu tamanho.

Para fazermos essa configuração inicial de sequência, basta colocarmos no setup().

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

## Percorrendo o array

Agora que temos uma sequência de LEDs, podemos muito bem mandar piscar cada um deles nessa sequência. Mas não iremos fazer isso do jeito que já vimos. Dessa vez, usaremos um laço de repetição:

void loop(){  
    for(int indice = 0; indice < TAMANHO\_SEQUENCIA; indice++){  
        piscaLed(sequenciaLuzes[indice]);  
    }  
}

O que fizemos? Dentro do nosso laço for, criamos a variável indice, que começará com o valor 0. Após cada interação, ela aumenta o seu valor em 1*(indice++)*. Dessa maneira, nós percorremos toda a nossa sequência de luzes, sem ter que ficar escrevendo diversas linhas de código. O melhor, é que se aumentarmos o tamanho da nossa sequência de LEDs, podemos continuar com o mesmo código.

No capítulo anterior aprendemos a piscar os LEDs de diversas formas. Neste capítulo, veremos como **ler** a sequência luminosa clicada pelo jogador, ou seja, veremos como entender qual foi a *resposta* dele. Para isso, nós utilizaremos os **botões**.

Lembra-se como procedemos com o LED? Nós primeiro fizemos um teste e apenas observamos para que depois pudéssemos reproduzir!

Nós vamos repetir essa estratégia, primeiro, analisaremos a lógica de um botão para depois expandir a conclusão para cada LED restante, ou seja, para os demais*botões*. Assim, enviamos um programa vazio para o **Arduino** e dessa maneira poderemos retirar o circuito sem preocupações. Fazendo isso nós podemos deixar **apenas** o circuito vermelho. Vamos no Fritzing , nele poderemos verificar como está o circuito.

Podemos realizar uma busca no Fritzing por button e selecionar o push button que é a versão de um botão simples com 4 pernas. Mas, não precisamos nos preocupar com ela, pois nós optaremos por utilizar uma versão que possui apenas 2 pernas, ou seja, uma de *entrada* e outra de *saída*.

Precisamos pensar em como comunicar para o **Arduino** que o botão foi pressionado. Primeiro, você precisa saber que o botão *corta* o circuito e quando pressionado ele se**fecha** permitindo, assim, a corrente passar.

## Identificando o clique no botão

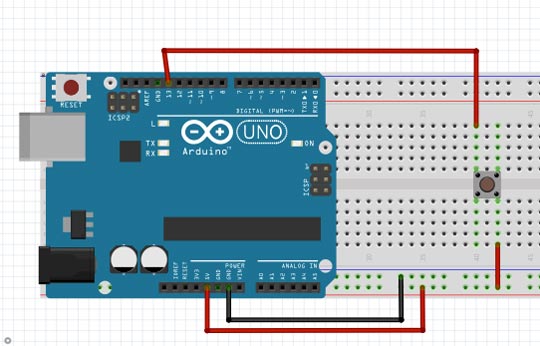
Como identificar que o botão foi pressionado? Podemos seguir o seguinte caminho:

Vamos conectar uma entrada de 5V no botão;

No seu pino de saída, nós ligamos um fio até uma das portas do **Arduino**;

Caso não chegue nenhuma corrente na porta, quer dizer que o botão está parado, sem ninguém mexendo nele. Porém, quando alguém pressioná-lo, o circuito irá se fechar e uma corrente de 5V chegará na porta escolhida, o que indicará que o botão foi pressionado.

E isso funciona! Observe como fica o circuito no Fritzing:



Agora, precisamos **ler** essa entrada 5V. Para fazer isso vamos iniciar um novo arquivo para testar nossas ideias! Quando elas estiverem melhor estabelecidas nós podemos passar os códigos para o programa principal e, assim, evitamos que fique confuso. Neste circuito, temos o LED\_VERMELHO conectado à porta **4** do arduino, e o BOTAO\_VERMELHOconectado à porta **10**. Configurando as portas teremos:

#define LED\_VERMELHO 4  
#define BOTAO\_VERMELHO 10  
  
  
void setup(){  
    pinMode(LED\_VERMELHO, OUTPUT);  
  
  
    pinMode(BOTAO\_VERMELHO, INPUT);  
  
  
    digitalWrite(LED\_VERMELHO, HIGH);  
  
  
    delay(1000);  
  
  
    digitalWrite(LED\_VERMELHO, LOW);  
}

Podemos configurar o LED\_VERMELHO normalmente e, ademais, adicionamos o BOTAO\_VERMELHO, que diferentemente dos nossos LEDs, é uma porta de **entrada** de sinal. Piscaremos a luz vermelha apenas para ter a certeza de que o programa está rodando corretamente. Tendo feito isso, salve o novo arquivo e envie para o **Arduino**. Agora, tudo deverá ocorrer corretamente!

Nesse momento, vamos utilizar o Serial para descobrir se o botão foi pressionado ou não. Por isso, no setup(), adicionamos a linha que configura o serial: Serial.begin(9600);.

No loop() tentaremos **ler** a entrada do botão. A dica já foi dada, da mesma maneira que enviamos um sinal utilizando o digitalWrite() vamos **ler** o sinal aplicando a função digitalRead(). Ao inserir isso no loop() poderemos verificar constantemente qual o sinal que chega na porta do botão. No caso, o **Arduino** identifica os sinais através do **1** ou **0**; o **0** indica “nenhum” e o número **1** mostra quando o sinal é receptado. Entretanto, para inspecionar isso não basta colarmos o valor do sinal no Serial:

void loop(){  
    int estadoBotao = digitalRead(BOTAO\_VERMELHO);  
    Serial.println(estadoBotao);  
    delay(500);  
}

Entretanto, compilando isso e enviando para o **Arduino** vemos que, infelizmente, não funciona! Ao inspecionarmos o Serial percebemos que nossa porta recebe constantemente o 0, e mesmo sem apertarmos o botão, recebe o sinal 1. Por que isso acontece?

## Veja o circuito abaixo:

## https://lh3.googleusercontent.com/HfhQC9DZWmrWBACMH_vrPonNe5tarUQPDKzGzjWVPZzCl3EVq5-m9PLS6cqbWYcz3Vyu5O8CzAzJZxnEnGap2rruvLxs2ORYQ2yXHYIUf6__XaZE6MeXkW49VLZ0dq14ETfssu6G

## No programa, nós configuramos a porta 13 da seguinte maneira:

## pinMode(13, INPUT\_PULLUP);

## E queremos ler essa porta 13 através da função digitalRead:

## int estado = digitalRead(13);

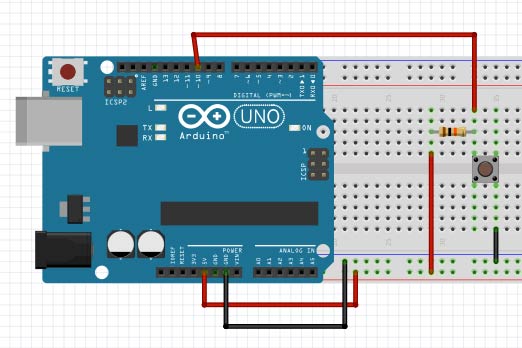
## O que podemos dizer ao tentar ler a porta 13?

No exemplo, significa que a porta está HIGH e que o botão **não** está pressionado. Esse é o comportamento quando configuramos a porta como INPUT\_PULLUP. A porta terá leitura LOW quando o botão for pressionado.

Para entender o que acontece com a estratégia adotada, podemos mudar a visualização do Fritzing de protoboard para Esquemático. Dessa forma, será mais fácil visualizar como o circuito funciona internamente. No caso, nosso botão é representado como uma espécie de interruptor. Perceba que existe uma corrente de 5V que não consegue atravessar o circuito, pois ele não está fechado devido a esse botão. Mas, e o outro lado? O que fica ligado com a porta 10? Qual será a corrente que está passando por ali? A resposta é **indefinida**.

Dependendo do local que você esteja, o ar pode ter *estática* e, por causa disso, uma corrente começa a passar pelo outro lado. Essa é a explicação do porquê de recebermos o sinal 1 sem ter pressionado o botão. Será que existe alguma maneira de evitarmos essa estática do ar? É claro que existe! Mas será preciso mudar nossa abordagem.

Nessa mudança, vamos alterar a ordem dos papéis. Em vez de ligarmos um fio de 5V no botão, faremos uma conexão com o GND, ou seja, o terra. Dessa maneira, esse lado agora irá *"absorver"* energia. Do outro lado, puxaremos um fio de 5V, e manteremos um outro fio conectado na porta 10.



## Entendendo o PULL-UP

Como irá funcionar? Quando o botão não estiver pressionado, estará chegando 5V na porta 10, ou seja, sinal **1**. Porém, quando pressionarmos o botão, o terra irá roubar todos os elétrons da corrente fazendo com que não chegue nada na porta 10 do **Arduino**, então, receberemos o sinal **0**.

**Importante**: perceba que colocamos uma resistência antes de chegar ao fio que conecta ao **Arduino**, pois não devemos conectar uma força diretamente nas portas do **Arduino**devido a possibilidade de sobrecorrente. Utilizamos um resistor bem potente, pois estamos pegando uma energia total de 5V, já para verificar o sinal, basta uma pequena corrente passando até a nossa porta. A nossa escolha foi por uma resistência de **10K ohm**.

O nome da técnica que utilizamos para resolver o problema é **pull-up**. Se testarmos o código com esse novo circuito veremos que tudo estará funcionando! No Serial, veremos o número 1 sendo escrito constantemente e quando apertamos o botão, receberemos o sinal 0.

Para facilitar nossa vida e não termos preocupações com os 4 resistores e mais os 4 fios em nossa protoboard, o **Arduino** já é preparado para conexões do tipo **pull-up**. Precisamos apenas informá-lo que ele mesmo conectará a porta num resistor interno de **10K ohm**. Para isso, no código de configuração da porta 10, vamos fazer a seguinte alteração:

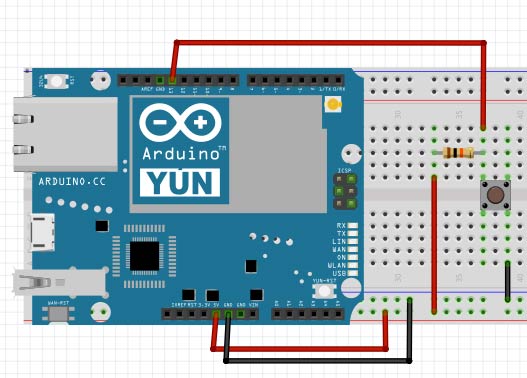
//Jeito antigo: pinMode(BOTAO\_VERMELHO, INPUT);  
pinMode(BOTAO\_VERMELHO, INPUT\_PULLUP);

Esse procedimento facilita muito a vida! Agora, temos um **pull-up** simulado pelo arduino!

Precisávamos dar toda essa volta para que você entedesse como nosso circuito está funcionando e não apenas “chutasse” sobre seu comportamento.

Nesse exercício vamos montar um circuito PULL-UP como apresentado no vídeo. Mas, **não** vamos utilizar a configuração PULL-UP do Arduino.

Primeiro, monte o circuito abaixo usando o Arduino. Repare que estamos manuseando um **resistor de 10kΩ**.



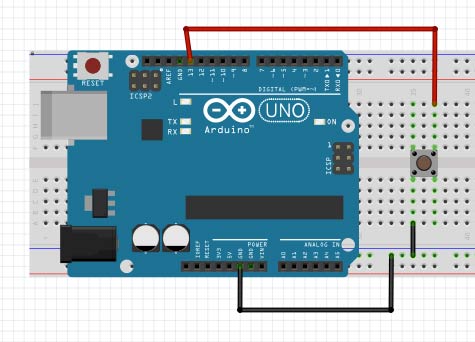
Uma vez montado, abra o ArduinoIDE e comece um novo *sketch* (novo programa). Na função setup, inicialize a porta 13 como INPUT e na função loop, leia o estado da porta:

void setup() {  
  Serial.begin(9600);  
  pinMode(13,INPUT);  
}  
  
void loop() {  
  int estado = digitalRead(13);  
  Serial.println(estado);  
  delay(500);  
}

Compile e carregue o seu programa. Fique de olho no monitor serial!

Apesar do Arduino já possuir esse resistor internamente, é interessante montar o circuito, pois isso ajuda no nosso aprendizado. Esse exercício possui o fim didático de praticar e ensinar, no entanto, para a montagem do jogo Genius usaremos os resistores internos do Arduino.

Agora, monte o circuito no seu Arduino utilizando um botão e usando a configuração INPUT\_PULLUP:



Repare que o botão tem uma ligação com o terra e, portanto, não é mais preciso usar um resistor externo. No ArduinoIDE crie um novo *sketch* e configure a porta como INPUT\_PULLUP:

void setup() {  
  Serial.begin(9600);  
  pinMode(13, INPUT\_PULLUP);  
}  
  
void loop() {  
  int estado = digitalRead(13);  
  Serial.println(estado);  
  delay(500);  
}

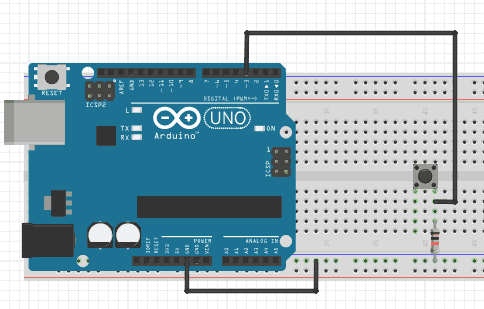
Compile e carregue o programa. Abra o monitor serial para ver o comportamento do seu botão!

Uma pergunta que pode surgir é a seguinte:

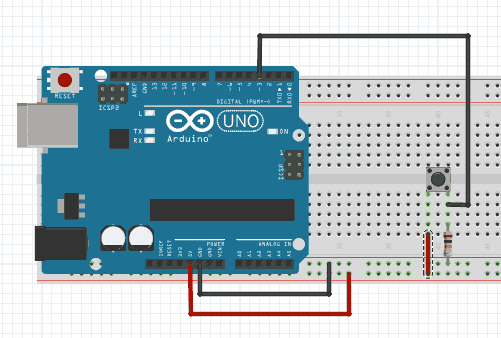
Já que podemos montar um circuito com estados definidos, utilizando a técnica PULL UP, não seria igualmente possível fazer um circuito com estados definidos utilizando uma forma inversa?

A resposta é **SIM**! O nome da técnica inversa é **Pull down** e, diferentemente do **Pull up** , **não podemos programar a porta de leitura** do botão para assumir este comportamento automaticamente. Então, é preciso montar este circuito, mas como ele seria?

Vamos pensar um pouco! Precisamos de uma ligação com o terra quando o botão não estiver apertado, assim recebemos LOW na nossa porta (na imagem abaixo usamos a porta 3):



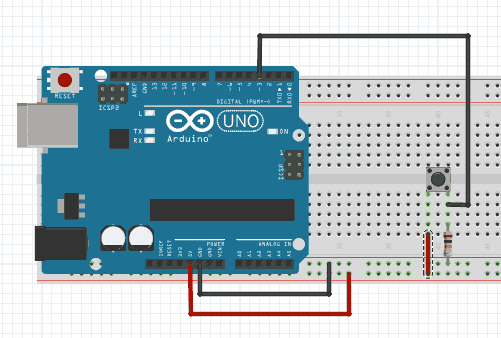
Repare que usamos um resistor de 10kΩ. Agora quando apertamos o botão devemos fechar o circuito entre a porta 3 e a porta de 5V do Arduino.



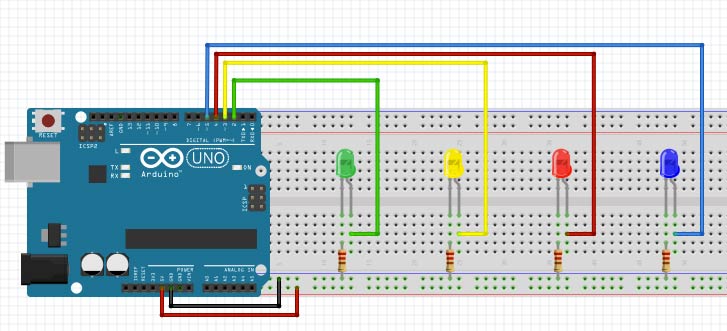
Como usamos um resistor de 10kΩ a energia vai fluir para porta 3.

Neste caso o botão não pressionado não deixa um circuito aberto uma vez que estamos ligados diretamente ao Terra através de um Resistor de 10KΩ. Assim, a leitura da porta é bem definida já que é garantido que será LOW/0.

Quando o botão é pressionado poderíamos imaginar que o Terra se encarregará de retirar toda a energia mas este mesmo resistor retarda esta ação permitindo que a energia alcance a porta do Arduino mudando seu estado para HIGH/1.

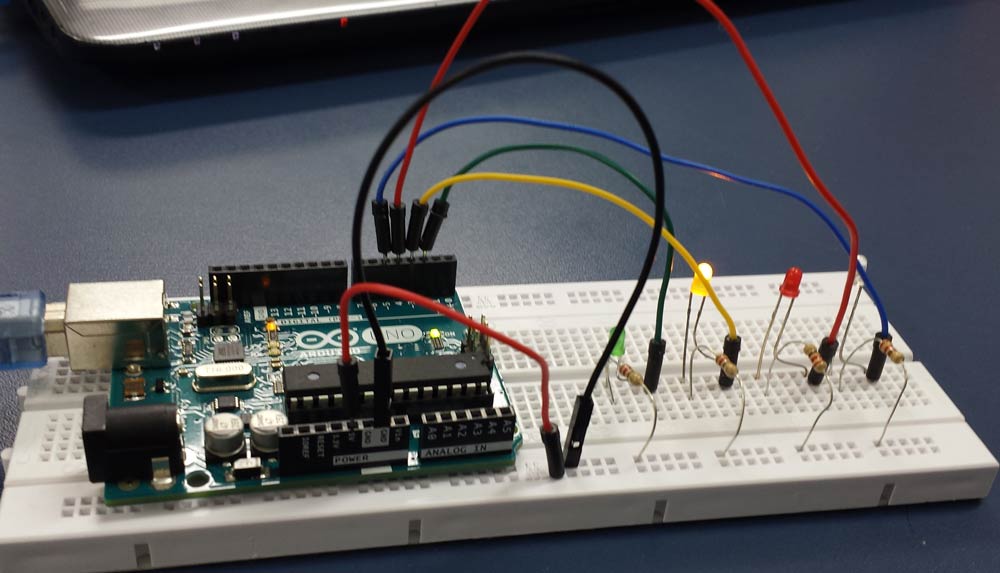


Agora que sabemos como funcionam os LEDs e o Botão precisamos repetir os mesmos procedimentos para todas as demais cores. Preste atenção, pois este capítulo serve para **organizar** a sua protoboard que pode ficar muito confusa ao longo do desenvolvimento do projeto. Começaremos ajeitando os 4 LEDs:

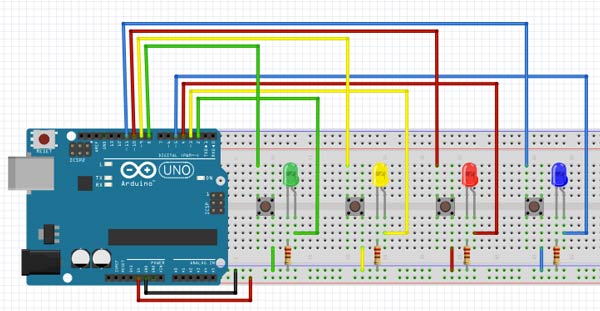


Lembre-se: Proteja os LEDs utilizando um resistor!

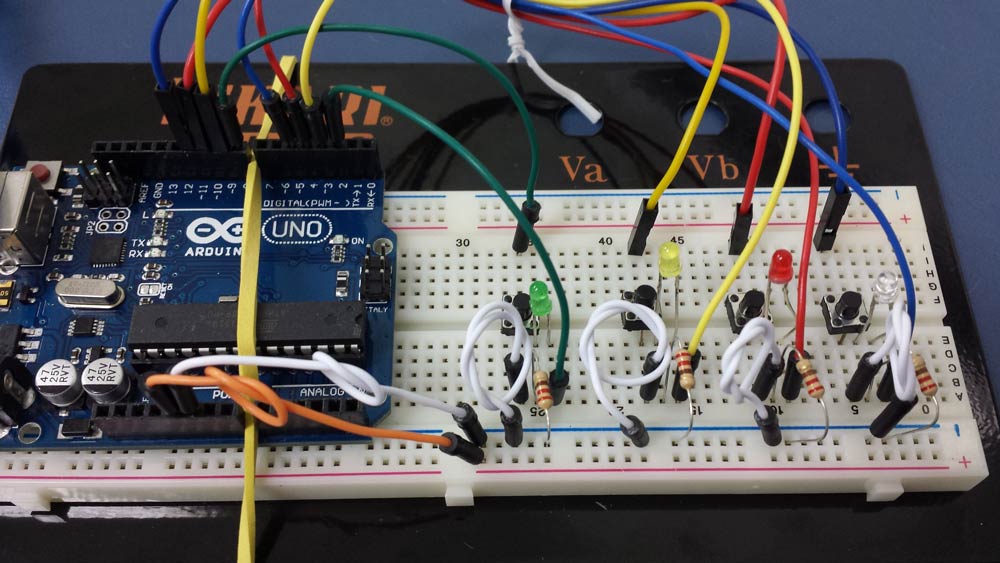
Aqui está a foto do circuito real:



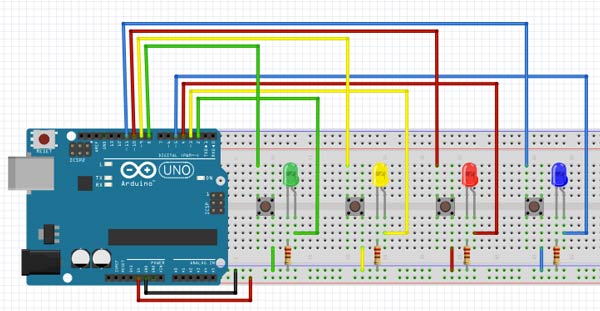
Uma vez montados os 4 LEDs com seus respectivos resistores, chega a hora de colocar os botões. Sempre um botão ao lado de cada LED. O resultado final deve ser parecido com o circuito abaixo:



Observe na imagem abaixo o circuito verdadeiro. Não se preocupe com a cor dos cabos! O importante é que ele esteja bem organizado. Uma dica é sempre enrolar os fios, para que não ocupem tanto espaço!



Uma vez montados os 4 LEDs com seus respectivos resistores, chegou a hora de inserir os botões! Sempre lembrando que cada botão deve ficar ao lado de um LED. O resultado final deve ficar parecido ao circuito abaixo:



Agora, está na hora de retornar ao código e passar o trecho referente ao botão para o código original do nosso jogo. No caso, perceba que temos a função iniciaPortas(). Por que não usar a mesma função para iniciar os botões? Primeiramente, declaramos nossos botões:

#define BOTAO\_VERDE 8

#define BOTAO\_AMARELO 9

#define BOTAO\_VERMELHO 10

#define BOTAO\_AZUL 11

E, assim, usamos o iniciaPortas():

void iniciaPortas(){

pinMode(LED\_VERDE, OUTPUT);

//outros LEDs

pinMode(BOTAO\_VERDE, INPUT\_PULLUP);

pinMode(BOTAO\_AMARELO, INPUT\_PULLUP);

pinMode(BOTAO\_VERMELHO, INPUT\_PULLUP);

pinMode(BOTAO\_AZUL, INPUT\_PULLUP);

}

Compilando o código e enviando para o **Arduino** vemos que não temos nada de errado nele e, inclusive, nossa sequência definida há algum tempo continua sendo repetida pelos LEDs.

## Lendo os botões

Já sabemos como ler um botão. Porém, como revisão, podemos adicionar dentro do nosso loop() o seguinte:

int estado = digitalRead(BOTAO\_VERDE);

Serial.println(estado);

E ao observar no monitor a resposta verificamos que recebemos o 1, mas quando o botão é pressionado, o sinal passa a ser 0. Ou seja, tudo funciona perfeitamente! Vamos apenas melhorar o que já temos. Primeiramente, vamos comentar o laço forque temos dentro do loop():

void loop(){

/\*

for(int indice = 0; indice < TAMANHO\_SEQUENCIA; indice++){

piscaLed(sequenciaLuzes[indice]);

}

\*/

int estado = digitalRead(BOTAO\_VERDE);

Serial.println(estado);

}

Mas, nós não queremos conhecer apenas o estado de **um** botão, queremos saber o de todos. Para isso, podemos criar uma função checaRespostaJogador(). Essa função irá nos devolver qual foi o botão apertado pelo usuário. Vamos começar a criar:

int checaRespostaJogador(){

if(digitalRead(BOTAO\_VERDE) == LOW){

return LED\_VERDE;

}

if(digitalRead(BOTAO\_AMARELO) == LOW){

return LED\_AMARELO;

}

if(digitalRead(BOTAO\_VERMELHO) == LOW){

return LED\_VERMELHO;

}

if(digitalRead(BOTAO\_AZUL) == LOW){

return LED\_AZUL;

}

}

Essa função nós leva até o if se o botão for apertado. Caso ele tenha sido pressionado a função retornará o LED correspondente. Mas e se e ninguém apertou o botão? Podemos retornar INDEFINIDO e, dessa maneira, o código fica expressivo e podemos verificar que ninguém clicou no botão, ou seja, o estado não está definido. Observe:

//resto do código

if(digitalRead(BOTAO\_AZUL) == LOW){

return LED\_AZUL;

}

return INDEFINIDO;

}

Ainda, precisamos definir a nossa constante. Ou seja, no início do programa é preciso atribuir um valor que não pode ser igual a algo comumente utilizado para mapear portas, por exemplo. Logo, não é uma boa ideia colocar 0, pois a atribuição de portas começa em 0. Tão pouco é aconselhável utilizar um número como 20, pois caso movêssemos o código para outro tipo de **Arduino** que tivesse 20 portas, nosso programa deixaria de funcionar. Bem, existe uma solução muito boa, que é -1. Esse é um número inteiro e nunca teremos uma porta mapeada como -1. Portanto:

#define INDEFINIDO -1

E dentro do loop() podemos apagar o código auxiliar e deixar apenas o comentário do laço. Depois disso, adicionamos uma saída ao nosso Serial, chamando a função. Dessa maneira, a tela ficará "printando" -1 até apertarmos algum botão. Observe como fica o código:

loop(){

//laço comentado

Serial.println(checaRespostaJogador());

}

## Uma última alteração

Para finalizar esse capítulo, podemos aprender a **piscar o LED correspondente**. E existe uma maneira muito fácil de arrumar isso! É um alívio perceber que o código está bem arrumado, pois isso torna as tarefas muito mais fáceis!

A função piscaLed() começará a devolver um inteiro que é o valor correspondente do LED. E na função checaRespostaJogador() deve ser retornada a função piscaLed(). Veja como fica o código:

int piscaLed(int portaLed){

digitalWrite(portaLed, HIGH);

delay(UM\_SEGUNDO);

digitalWrite(portaLed, LOW);

delay(MEIO\_SEGUNDO);

return portaLed;

}

int checaRespostaJogador(){

if(digitalRead(BOTAO\_VERDE) == LOW){

return piscaLed(LED\_VERDE);

}

if(digitalRead(BOTAO\_AMARELO) == LOW){

return piscaLed(LED\_AMARELO);

}

if(digitalRead(BOTAO\_VERMELHO) == LOW){

return piscaLed(LED\_VERMELHO);

}

if(digitalRead(BOTAO\_AZUL) == LOW){

return piscaLed(LED\_AZUL);

}

return INDEFINIDO;

}

Compilando e enviando poderemos verificar que isso funciona! Nossos LEDs acendem conforme clicamos nos botões!

Conforme visto no vídeo, é preciso verificar se um botão foi pressionado. Para tanto, precisamos de uma nova constante que representa que nenhum botão foi apertado.

No início do programa crie uma nova constante, com o nome INDEFINIDO:

#define INDEFINIDO -1

Depois disso, logo abaixo da função loop, adicione uma nova função, a checaRespostaJogador, que serve para verificar o estado das portas que ficam conectadas aos botões:

int checaRespostaJogador() {

if(digitalRead(BOTAO\_VERDE) == LOW) {

return LED\_VERDE;

}

if(digitalRead(BOTAO\_AMARELO) == LOW) {

return LED\_AMARELO;

}

if(digitalRead(BOTAO\_VERMELHO) == LOW) {

return LED\_VERMELHO;

}

if(digitalRead(BOTAO\_AZUL) == LOW) {

return LED\_AZUL;

}

return INDEFINIDO;

}

Altere a função loop para usar checaRespostaJogador:

void loop() {

for(int indice = 0; indice < TAMANHO\_SEQUENCIA; indice++){

piscaLed(sequenciaLuzes[indice]);

}

int estadoApertado = checaRespostaJogador();

Serial.println(estadoApertado);

}

Compile o programa e carregue no Arduino. Aperte alguns botões e verifique o monitor serial!

Agora, só falta ascender o LED correspondente e essa será tarefa do próximo exercício.

Para que você acompanhe de maneira mais fácil, segue o código completo:

//int ledVermelho = 2;

#define LED\_VERDE 2

#define LED\_AMARELO 3

#define LED\_VERMELHO 4

#define LED\_AZUL 5

#define INDEFINIDO -1

#define BOTAO\_VERDE 8

#define BOTAO\_AMARELO 9

#define BOTAO\_VERMELHO 10

#define BOTAO\_AZUL 11

#define UM\_SEGUNDO 1000

#define MEIO\_SEGUNDO 500

#define TAMANHO\_SEQUENCIA 4

int sequenciaLuzes[TAMANHO\_SEQUENCIA];

void setup() {

// put your setup code here, to run once:

Serial.begin(9600);

iniciaPortas();

iniciaJogo();

}

void iniciaJogo(){

sequenciaLuzes[0] = LED\_AZUL;

sequenciaLuzes[1] = LED\_VERDE;

sequenciaLuzes[2] = LED\_VERMELHO;

sequenciaLuzes[3] = LED\_AMARELO;

}

void iniciaPortas(){

pinMode(LED\_VERDE,OUTPUT);

pinMode(LED\_AMARELO,OUTPUT);

pinMode(LED\_VERMELHO,OUTPUT);

pinMode(LED\_AZUL,OUTPUT);

pinMode(BOTAO\_VERDE,INPUT\_PULLUP);

pinMode(BOTAO\_AMARELO,INPUT\_PULLUP);

pinMode(BOTAO\_VERMELHO,INPUT\_PULLUP);

pinMode(BOTAO\_AZUL,INPUT\_PULLUP);

}

void loop() {

for(int indice = 0; indice < TAMANHO\_SEQUENCIA; indice++){

piscaLed(sequenciaLuzes[indice]);

}

int estadoApertado = checaRespostaJogador();

Serial.println(estadoApertado);

}

int checaRespostaJogador() {

if(digitalRead(BOTAO\_VERDE) == LOW) {

return LED\_VERDE;

}

if(digitalRead(BOTAO\_AMARELO) == LOW) {

return LED\_AMARELO;

}

if(digitalRead(BOTAO\_VERMELHO) == LOW) {

return LED\_VERMELHO;

}

if(digitalRead(BOTAO\_AZUL) == LOW) {

return LED\_AZUL;

}

return INDEFINIDO;

}

void piscaSequencia1(){

piscaLed(LED\_VERDE);

piscaLed(LED\_AMARELO);

piscaLed(LED\_VERMELHO);

piscaLed(LED\_AZUL);

delay(MEIO\_SEGUNDO);

}

void piscaSequencia2(){

digitalWrite(LED\_VERDE,HIGH);

digitalWrite(LED\_AMARELO,HIGH);

digitalWrite(LED\_VERMELHO,HIGH);

digitalWrite(LED\_AZUL,HIGH);

delay(UM\_SEGUNDO);

digitalWrite(LED\_VERDE,LOW);

digitalWrite(LED\_AMARELO,LOW);

digitalWrite(LED\_VERMELHO,LOW);

digitalWrite(LED\_AZUL,LOW);

delay(MEIO\_SEGUNDO);

}

int piscaLed(int portaLed){

digitalWrite(portaLed,HIGH);

delay(UM\_SEGUNDO);

digitalWrite(portaLed,LOW);

delay(MEIO\_SEGUNDO);

return portaLed;

}

Conforme vimos no vídeo, procure a função piscaLed no seu código. Altere essa função para devolver a portaLed:

int piscaLed(int portaLed){

digitalWrite(portaLed,HIGH);

delay(UM\_SEGUNDO);

digitalWrite(portaLed,LOW);

delay(MEIO\_SEGUNDO);

return portaLed;

}

Agora, chame a função piscaLed na função checaRespostaJogador:

int checaRespostaJogador() {

if(digitalRead(BOTAO\_VERDE) == LOW) {

return piscaLed(LED\_VERDE);

}

if(digitalRead(BOTAO\_AMARELO) == LOW) {

return piscaLed(LED\_AMARELO);

}

if(digitalRead(BOTAO\_VERMELHO) == LOW) {

return piscaLed(LED\_VERMELHO);

}

if(digitalRead(BOTAO\_AZUL) == LOW) {

return piscaLed(LED\_AZUL);

}

return INDEFINIDO;

}

Compile e teste o seu código!

Vamos olhar com atenção a função iniciaJogo(). Ela está da seguinte maneira:

void iniciaJogo() {  
    sequenciaLuzes[0] = LED\_AZUL;  
    sequenciaLuzes[1] = LED\_VERDE;  
    sequenciaLuzes[2] = LED\_VERMELHO;  
    sequenciaLuzes[3] = LED\_AMARELO;  
}

Com esse formato nós sempre teremos a mesma sequência de LEDs acendendo e esse não é o objetivo do jogo! Para resolver esse problema vamos utilizar uma função que gere um número randômico. Podemos acrescentar int randomico = random(1, 10). Atenção, quando escrevemos dessa maneira temos um intervalo de números de 1 até 9, o número 1 é incluído, mas o 10 é o limite. Caso quiséssemos um intervalo de dez números, deveríamos escrever (1, 11). Com o auxílio do monitor serial podemos verificar se isso funciona. Teremos o seguinte código:

void iniciaJogo() {  
    int randomico = random(1,11);//1-10  
    Serial.println(randomico);  
    sequenciaLuzes[0] = LED\_AZUL;  
    sequenciaLuzes[1] = LED\_VERDE;  
    sequenciaLuzes[2] = LED\_VERMELHO;  
    sequenciaLuzes[3] = LED\_AMARELO;  
}

Na função loop nós podemos retirar o Serial.println(checaRespostaJogador);escrevendo duas barras na frente, //, ou seja, apenas inserindo um comentário. Agora, podemos compilar isso e enviar. Abrindo o Monitor verificamos que ele gera o número 8. Vamos apagar o que tínhamos escrito:

int randomico = random(1,11);//1-10  
    Serial.println(randomico);

Acrescentamos que a primeira posição deve ser um número randômico que vai variar das posições do LED verde até o azul. Portanto, escrevemos que sequenciaLuzes[0] = random(LED\_VERDE, LED\_AZUL + 1). Observe que adicionamos o +1, pois dessa maneira, temos certeza de que o LED azul estará incluído no intervalo. E repetimos isso para os demais LEDS:

void iniciaJogo() {  
  
    sequenciaLuzes[0] = random(LED\_VERDE, LED\_AZUL + 1);//2-5 (6)  
    sequenciaLuzes[1] = random(LED\_VERDE, LED\_AZUL + 1)  
    sequenciaLuzes[2] = random(LED\_VERDE, LED\_AZUL + 1)  
    sequenciaLuzes[3] = random(LED\_VERDE, LED\_AZUL + 1)  
}

Mas, atenção! A todo momento estivemos em busca de um código mais compacto e organizado possível, portanto, vamos criar uma função que sorteie a cor! Ela será a int sorteiaCor(). Vamos adicionar um return random (LED\_VERDE, LED\_AZUL +1) e utilizamos essa função junto ao iniciaJogo(). Podemos realizar uma construção usando o for! Teremos for(int indice = 0; indice < TAMANHO\_SEQUENCIA; indice ++). O comentário //2-5 (6) nós deslocaremos para junto do sorteiaCor. Utilizando isso o código fica ainda mais compacto! Observe:

sequenciaLuzes[indice] = sorteiaCor();

Teremos o seguinte

void iniciaJogo() {  
    for(int indice = 0; indice < TAMANHO\_SEQUENCIA; indice ++) {  
    sequenciaLuzes[indice] = sorteiaCor();  
    }  
}  
  
int sorteiaCor() {  
    return random(LED\_VERDE, LED\_AZUL + 1); // 2-5 (6)  
}

Agora, podemos sortear o jogo! Para saber qual jogo está saindo, vamos tirar o comentário /\* do void loop. Assim, poderemos verificar a sequência sendo gerada randomicamente!

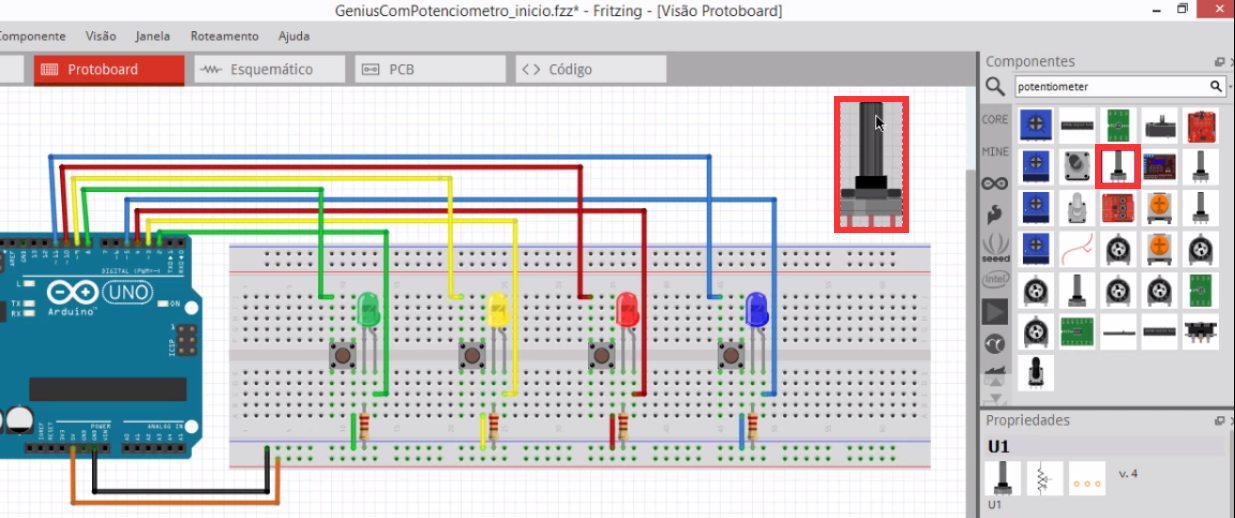
Nós compilamos isso e enviamos para o **Arduino** que irá iniciar uma sequência gerada randomicamente! Mas, ele mostra uma sequência que se repetirá, mesmo que nós apertemos o "Reset" a sequência será a mesma. Ou seja, a função random() não está funcionando.

Vamos entender o que aconteceu! Um computador só sabe fazer funções matemáticas através de circuito e ele usa isso para fazer praticamente tudo, por exemplo, gerar vídeos, áudio, receber e-mails, etc. Dessa maneira, quando falamos para o computador que ele deve sortear algo e que isso deve ser aleatório é preciso que nós alimentemos essa função com algo que seja igualmente aleatório.

Normalmente, o sistema operatório possui um relógio que é utilizado para alimentar a função que gera um número aleatório. O relógio é representado por números que são convertidos em uma determinada hora e quando o jogo começa nós não sabemos que número é esse.

O grande problema, entretanto, é que nós não temos um relógio no **Arduino**. E isso nos causa problemas, portanto, teremos que ser criativos para lidar com a situação. E se pudéssemos controlar o número que estamos passando para função? E toda vez que mudássemos esse número, nós trocássemos o jogo? Uma maneira de fazer isso é utilizar um número que seja lógico. Como podemos fazer isso?

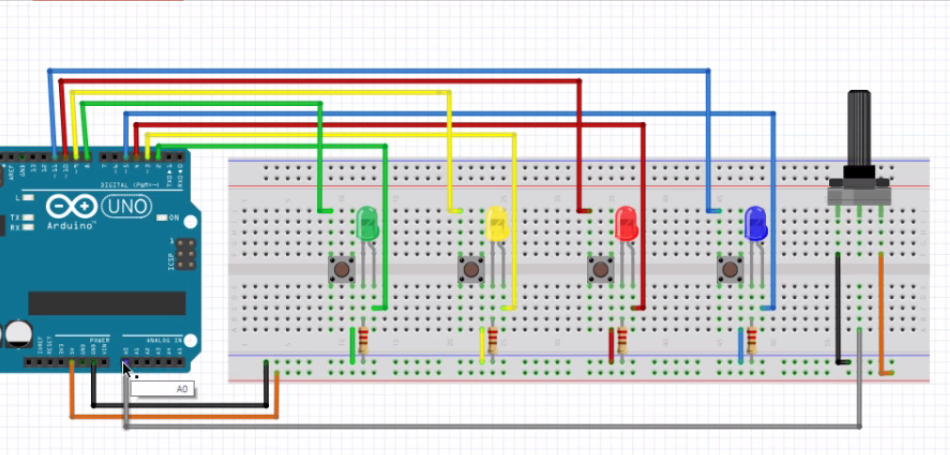
Certamente, não podemos fazer isso utilizando uma porta lógica, pois ela aceita apenas 2 valores, ou 0 ou 1. Dessa forma, vamos utilizar uma porta analógica que pega o sinal contínuo do ambiente. Por exemplo, poderíamos colocar um sensor de temperatura, pressão e esse número contínuo seria pego e convertido em um número inteiro. Assim, podemos criar projetos utilizando esses artefatos! O *potenciômetro* é muito interessante, pois é uma peça simples e fácil de ser utilizada. Observe o potenciômetro no **Fritzing**:



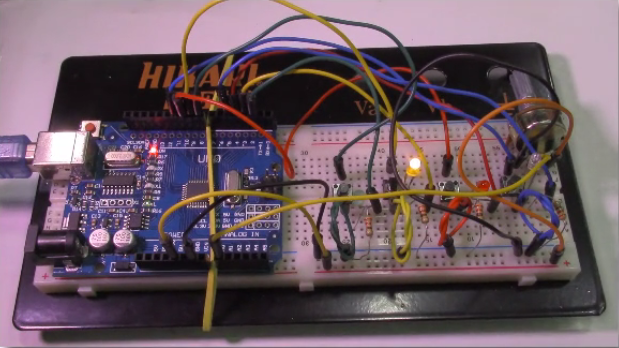
O potenciômetro possui uma resistência variável o que significa que somos nós que dizemos qual é a sua resistência. Assim, ao rodá-lo nós podemos passar instruções para que fique com mais ou menos resistência. Isso é legal, pois, o sinal resultante da resistência pode gerar nosso número.

Para fazer isso é preciso, primeiro, aprender como o potenciômetro funciona. Existe a força, na linha de baixo e o terra na linha de cima. O que faremos é ligar os dois sinais nas portas externas do potenciômetro. O primeiro fio que ligaremos será o terra, em preto, e o outro será o força, que ficará na cor laranja. A porta que sobra é a que fornece o sinal resultante, significa que o potenciômetro vai receber nessa entrada 5v e através dessa regulagem conseguiremos um sinal menor se aumentarmos a resistência ou maior se a diminuirmos. Então, vamos alimentar esse sinal no **Arduino**. Vamos ligá-lo, por exemplo, a porta 0.

Resumindo, vamos ligar uma porta exterior na força (laranja) e a outra porta exterior no terra (preto) e o do meio (cinza) será o sinal resultante que nós conectamos a porta analógica, ou seja, estamos ligando na porta 0. Observe:



Nós já vimos o processo de montagem do circuito através do **Fritzing**, agora, podemos construí-lo na mão:



Após criarmos o circuito usando o potenciômetro, vamos colocar ele para funcionar e faremos isso através de um sinal analógico. Na função iniciaJogo(), como queremos ver o potenciômetro em ação, vamos inserir o int potencimetro para fazer uma leitura dele, essa função é similar a analogRead(0). Observe que O zero equivale a porta analógica. Feito isso, vamos voltar a usar o monitor serial e observar qual é o estado que está sendo gerado, assim, escrevemos Serial.println(potenciometro). Teremos o seguinte:

void iniciaJogo() {  
  int potenciometro = analogRead(0);  
  Serial.println(potenciometro);  
  for(int indice = 0; indice < TAMANHO\_SEQUENCIA; indice ++){  
    sequenciaLuzes(indice) = sorteiaCor();  
  }  
}

Compilando isso e abrindo o monitor serial vemos que o valor que recebemos é de 562! Se alteramos o potenciômetro girando ele, teremos um resultado distinto e isso acontecerá sempre que nós mexermos nesse componente. Além disso, podemos descobrir também quais são os resultados mínimos e máximos envolvidos! O número mínimo é 0 e o máximo é 1023. Ou seja, existe um intervalo de 1024 jogos possíveis! Esse é um número bastante razoável já que é muito difícil que um jogador memorize todas essas possibilidades!

Depois de ver o potenciômetro em ação podemos modificar o int potenciometro para int jogo. Entretanto, ler o potenciômetro ainda não foi o suficiente. É preciso gerar a função que lê os números aleatórios! Vamos fazer isso acrescentando no iniciaJogo()o randomSeed(jogo)

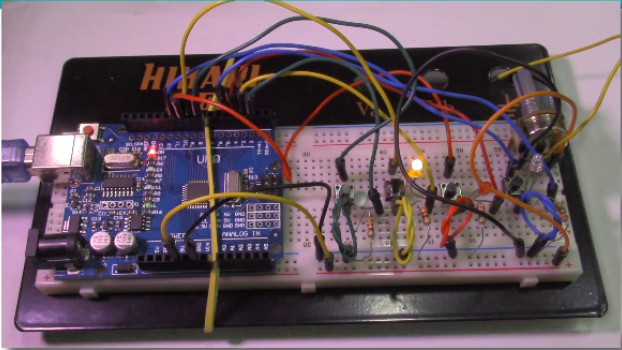
void iniciaJogo() {  
  int jogo = analogRead(0);  
  randomSeed(jogo);  
  Serial.println(potenciometro);  
  for(int indice = 0; indice < TAMANHO\_SEQUENCIA; indice ++){  
    sequenciaLuzes(indice) = sorteiaCor();  
  }  
}

Podemos salvar isso e, agora, toda vez que o jogo for resetado e o potenciômetro alterado nós teremos uma sequência diferente!

Existe, ainda, um segredo interessante! Não é preciso do potenciômetro para gerar uma sequência aleatória. Você se lembra da discussão sobre PULL UP? Nós comentamos sobre o circuito indefinido! O ar possui uma voltagem que não conseguimos delimitar o valor exato, mas que causa interferência, portanto, a carga elétrica variável do ar gera um número aleatório em nosso jogo.

Para testar isso, basta retirar a conexão com a porta analógica e um novo jogo será gerado! Assim, dando um "Reset", observamos uma sequência, ao dar um novo "Reset", observaremos uma segunda sequência distinta! Ou seja, funciona! Mas, nós não teremos mais controle sobre o jogo como tínhamos com o potenciômetro.

Resumindo, com a porta aberta e em contato com o ar, nós não temos nenhum controle, mas como o ar tem variações sutis em sua corrente, a leitura da porta aberta gera versões diferentes do jogo! Para chegar nesse ponto, tivemos que entender o que o **Arduino** faz para gerar um número aleatório, assim, toda essa explicação serviu para compreendermos o que estava acontecendo!



Aprendemos que uma boa maneira de obter uma fonte de aleatoriedade no Arduino é utilizar a leitura de uma porta analógica que esteja vazia. Isso sempre gera um número aleatório, por quê?

... o Arduino tenta interpretar a voltagem do ar, que tem uma carga elétrica variável e que depende de quase infinitos fatores, gerando assim um valor aleatório.

Em computadores tradicionais nós utilizamos o relógio do computador para gerar um número aleatório para a função random(). O Arduino, entretanto, não possui um relógio interno, portanto, aproveitamos as portas analógicas para obter um valor aleatório!

Ele faz isso tentando **interpretar o valor da voltagem do ar**, que possui uma carga elétrica variável e que depende de fatores como umidade, pressão, íons do ar, etc. E como são quase infinitos os fatores envolvidos e como eles mudam o tempo todo, o **Arduino**, na prática, acaba lendo um valor diferente toda vez.

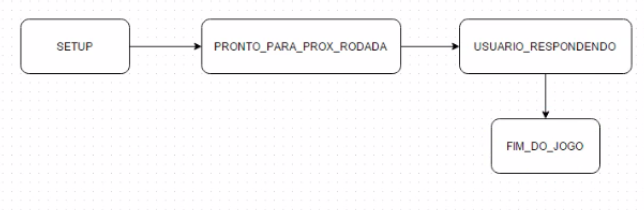
Primeiro, vamos retirar do circuito o potenciômetro, assim como desconectamos os fios terra e da força.

Vamos acessar o Draw io, um programa que nos auxilia através do recurso visual a planejar nosso projeto. Já temos cada peça necessária para o circuito, mas um segredo também é pensar no relacionamento dos componentes entre si! Esse programa nos ajuda de uma maneira fácil a chegar no resultado pretendido.

O primeiro passo é entender que o jogo passa por diversos estados. E o draw.io nos ajuda a visualizar a sequência desses acontecimentos. Vamos lá?!

1. O primeiro estado é facilmente identificável, o SETUP, ou, estado inicial. Portanto, criamos uma caixa equivalente para ele. Nesse estado utilizamos uma função que nos ajuda a iniciar portas, o jogo, etc.
2. Uma vez que o SETUP estiver pronto, podemos começar a jogar. Dessa maneira, estamos prontos para iniciar uma próxima rodada e é exatamente isso que vamos colocar na segunda caixa: PRONTO\_PARA\_PROX\_RODADA
3. Depois dessa rodada ele executa a função que mostra as luzes. Após a primeira sequência de luzes respondida, ocorrerá uma segunda, terceira, até que o usuário acerte todas. Nesse estado é bastante claro que o usuário está respondendo, por isso, colocamos uma terceira caixa USUARIO\_RESPONDENDO.
4. Podemos acrescentar setas que indicam o caminho do projeto. Portanto, vamos do SETUP para o PRONTO\_PARA\_PROX\_RODADA e, por fim, para o USUÁRIO\_RESPONDENDO.
5. O próximo estado é FIM\_DO\_JOGO. Assim, ligamos a seta do USUARIO\_RESPONDENDO a essa última caixa.

Por enquanto temos o seguinte:

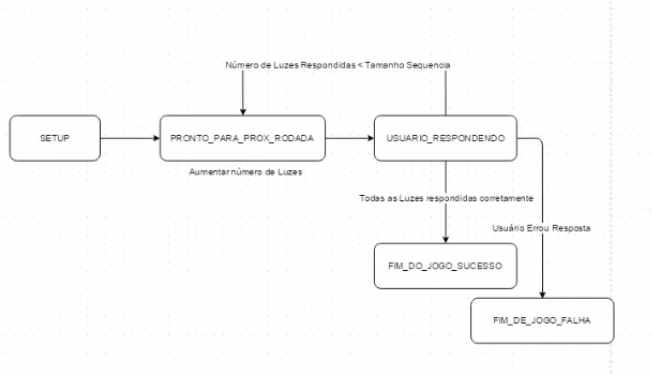


Mas, o projeto real não é tão perfeito! Esse esquema indica que nós temos apenas uma rodada e, na verdade, é um pouco mais complicado do que isso! Por exemplo, só é fim de jogo depois que o usuário finaliza todas as etapas, então, podemos acrescentar antes de FIM\_DO\_JOGO um comentário dizendo que "Todas as Luzes respondidas corretamente" indicam o fim do jogo.

Se as luzes não foram respondidas corretamente, recomeçam os processos, dessa maneira, nós inserimos uma seta que leva de volta de USUARIO\_RESPONDENDO para PRONTO\_PARA\_PROX\_RODADA e inserimos o comentário Número de Luzes Respondidas < Tamanho Sequencia. Assim, se o número de luzes for menor que a sequência, então, somos direcionados para o PRONTO\_PARA\_PROXIMA RODADA que deve possuir uma condição marcada pelo comentário: Aumentar número de luzes.

Esse é o fluxo do jogo até que o usuário consiga finalizar! Mas, não temos só uma maneira de finalização! Ele pode acabar de duas maneiras: FIM\_DO\_JOGO\_SUCESSO e FIM\_DE\_JOGO\_COM\_FALHA. Nessa segunda maneira nós acrescentamos um comentário, Usuário errou.

Com isso, temo um diagrama bastante fiel do que acontece no jogo:



Se você ainda estiver utilizando o potenciômetro, por favor, remova-o do circuito!

Ele já não é mais necessário, pois, estamos usando uma porta analógica aberta para gerar o *seed* do jogo.

Voltando ao código fonte vamos pensar em como identificar os diferentes estados! O primeiro nós já conhecemos, o SETUP, que está bem marcado. O segredo é no loop()entender a como mapear esses estados!

Em cada situação distinta o jogo se comportará de uma maneira diferente e isso pode causar muita confusão, já que o código pode ficar bem complicado. Por enquanto, ele ainda está pequeno pois, não está executando muitas tarefas. O que vamos fazer, primeiro, é separar a função do for que está junto ao loop(), pois isso é algo que vai acontecer várias vezes. Por isso, vamos acrescentar uma nova função tocaLedsRodada()e nela colocaremos o for:

void tocaLedsRodada(){  
    for(int indice = 0; indice < TAMANHO SEQUENCIA; indice++){  
        piscaLed(sequenciaLuzes[indice]);  
    }  
}

Feito isso, vamos aproveitar para remover o comentário do loop(). O primeiro objetivo é deixar o código expressivo, assim, seria ótimo se conseguíssemos marcar no código fonte que estamos percorrendo os diferentes estados. Como queremos um conjunto de valores para expressar uma determinada situação podemos declarar, usando a linguagem C, uma enumeração, portanto, escolhemos o enum Estados. Nós utilizaremos isso para informar quais foram os valores identificados, no caso: PRONTO\_PARA\_PROX\_RODADA, USUARIO\_RESPONDENDO, JOGO\_FINALIZADO\_SUCESSO e JOGO\_FINALIZADO\_FALHA. Teremos o seguinte:

enum Estados{  
  PRONTO\_PARA\_PROX\_RODADA,  
  USUARIO\_RESPONDENDO, JOGO\_FINALIZADO\_SUCESSO,  
  JOGO\_FINALIZADO\_FALHA  
};

É importante notar que a linguagem C trabalha de uma maneira bastante peculiar. Na verdade já vimos esse comportamento antes, quando o C usou um apelido para lidar com os diferentes LEDS. Quando usamos o enum estamos aplicando uma sequência numérica para estes itens, por exemplo, o PRONTO\_PARA\_PROX\_RODADA é o item número 0. Temos, portanto uma numeração que vai até 3. E, isso nos permite usar coisas como:

void loop() {  
    int estado = PRONTO\_PARA\_PROX\_RODADA;  
}

Compilando isso vemos que funciona! Ou seja, o PRONTO\_PARA\_PROX\_RODADA também é um apelido! Assim caso quiséssemos utilizar uma função para conhecer o estadoAtual() poderíamos utilizar o apelido dele e pedir para que o PRONTO\_PARA\_PROX\_RODADA fosse retornado, assim, na função loop() poderíamos especificar que int estado = estadoAtual. E incluiremos o comentário //Computar Estado Atual, deixaremos isso para discutir mais tarde. Teremos o seguinte:

void loop() {  
  int estado = estadoAtual();  
}  
  
int estadoAtual(){  
  //Computar Estado Atual  
  return PRONTO\_PARA\_PROX\_RODADA;  
}

É preciso compreender que a função loop estará rodando indefinidamente, então, dependendo do estado que o jogo estiver, vamos ter que mostrar algo ou ler algum botão ou dizer que um jogador ganhou ou perdeu. E como faremos isso sem ter um código que acabe ficando super confuso e complicado!?

Podemos, na função loop() acrescentar um switch e também um case. Assim, sinalizamos que caso tudo esteja pronto para a próxima rodada, os próximos passos podem ter seguimento, case PRONTO\_PARA\_PROX\_RODADA e inserimos o comentário //executa funções próxima rodada. Podemos ainda, completar afirmando o que ele fará em cada situação, assim, case USUARIO\_RESPONDENDO, case JOGO\_FINALIZADO\_SUCESSO e casa JOGO\_FINALIZADO\_FALHA. Teremos o seguinte:

void loop() {  
  switch(estadoAtual()) {  
  case PRONTO\_PARA\_PROX\_RODADA:  
    //executa funções próxima rodada  
  case USUARIO\_RESPONDENDO:  
  case JOGO\_FINALIZADO\_SUCESSO:  
  case JOGO\_FINALIZADO\_FALHA:  
  }  
}

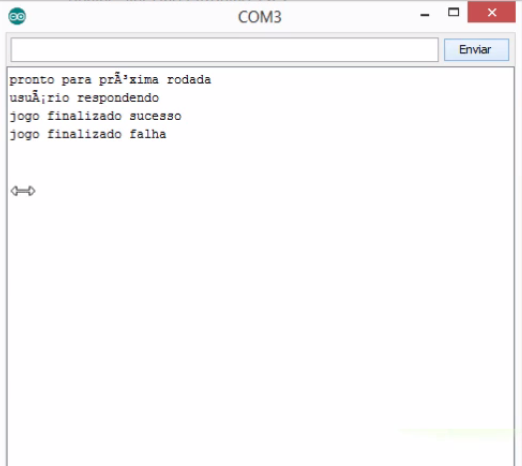
Tendo esse código o que acontece é que as tarefas serão executadas na ordem como estão escritas, de cima para baixo. Vamos ver funcionando? Escrevemos Serial.println(" ") acompanhado do respectivo componente:

void loop() {  
  switch(estadoAtual()) {  
  case PRONTO\_PARA\_PROX\_RODADA:  
    //executa funções próxima rodada  
    Serial.println("pronto para próxima rodada");  
  case USUARIO\_RESPONDENDO:  
    Serial.println("usuário respondendo");  
  case JOGO\_FINALIZADO\_SUCESSO:  
    Serial.println("jogo finalizado sucesso");  
  case JOGO\_FINALIZADO\_FALHA:  
    Serial.println("jogo finalizado falha");  
  }  
}

Compilamos isso e enviamos para o **Arduino**. Na verdade, estamos interessados em observar o **Console**, assim, vamos pedir para ele rodar, mas ele ficará travado em um mesmo lugar. Para fazer isso podemos utilizar a instrução for passando que ele deve ser nada, que checa nada e faz nada. Nós escrevemos abaixo do loop():

for(;;);

Compilando isso e enviando teremos o seguinte:



Tirando um pequeno problema de acentuação, tudo foi rodado! Mas, isso não vai nos ajudar, pois queremos que ele rode apenas uma coisa toda vez que o estado for checado! É preciso que ele cheque o estado, descubra que está pronto e execute a função que prepara a próxima rodada, depois queremos que ele saiba que é a hora do usuário responder e só execute a função das funções e que ele bote o finalizado com sucesso! Note que o resultado que obtemos depois do JOGO\_FINALIZADO\_SUCESSOé JOGO\_FINALIZADO\_FALHA. Ou seja, é preciso arrumar a função case.

Portanto, vamos apagar o comentário que havia sido inserido e vamos acrescentar um break que indica que nós quebramos a execução. Observe:

void loop() {  
  switch(estadoAtual()) {  
  case PRONTO\_PARA\_PROX\_RODADA:  
    Serial.println("pronto para próxima rodada");  
    break;  
  case USUARIO\_RESPONDENDO:  
    Serial.println("usuário respondendo");  
  case JOGO\_FINALIZADO\_SUCESSO:  
    Serial.println("jogo finalizado sucesso");  
  case JOGO\_FINALIZADO\_FALHA:  
    Serial.println("jogo finalizado falha");  
  }  
}

Compilando e mandando isso teremos o seguinte resultado:



Ou seja, só uma instrução foi executada! Assim se não há o break as demais instruções são executadas. Portanto, vamos acrescentar o break aos demais pontos:

void loop() {  
  switch(estadoAtual()) {  
  case PRONTO\_PARA\_PROX\_RODADA:  
    Serial.println("pronto para próxima rodada");  
    break;  
  case USUARIO\_RESPONDENDO:  
    Serial.println("usuário respondendo");  
    break;  
  case JOGO\_FINALIZADO\_SUCESSO:  
    Serial.println("jogo finalizado sucesso");  
    break;  
  case JOGO\_FINALIZADO\_FALHA:  
    Serial.println("jogo finalizado falha");  
    break;  
  }  
  for(;;);  
}

E isso terá resolvido o problema!

No início do programa adicione a enumeração, pois ela ajudará a identificar os estados do jogo.

enum Estados {  
  PRONTO\_PARA\_PROXIMA\_RODADA,  
  USUARIO\_RESPONDENDO,  
  JOGO\_FINALIZADO\_SUCESSO,  
  JOGO\_FINALIZADO\_FALHA  
};

Lembrando que cada valor da enumeração tem um inteiro associado, começando com o número 0.

Vamos escrever uma nova função, a estadoAtual. Ela vai descobrir em cada *loop* qual é estado atual do jogo.

Por enquanto, essa função só conhece o estado **PRONTO\_PARA\_PROXIMA\_RODADA**:

int estadoAtual() {  
      return PRONTO\_PARA\_PROXIMA\_RODADA;   
}

Vamos chamar esse função dentro do loop já aplicando a estrutura de controle de fluxo switch:

void loop() {  
  switch (estadoAtual()) {  
    case PRONTO\_PARA\_PROXIMA\_RODADA:  
      Serial.println("Pronto para a proxima rodada");  
      break;  
    case USUARIO\_RESPONDENDO:  
      Serial.println("Jogador respondendo");  
      break;  
    case JOGO\_FINALIZADO\_SUCESSO:  
      Serial.println("Jogo finalizado com sucesso");  
      break;  
    case JOGO\_FINALIZADO\_FALHA:  
      Serial.println("Jogo finalizado com falha");  
      break;  
  }  
  delay(MEIO\_SEGUNDO);  
}

Vamos retornar ao diagrama e compreender como mapear a transição. Nesse esquema temos diversas informações e uma grande parte delas já escrevemos no código fonte! É preciso ver o que ainda falta.

Por exemplo, o tamanho da sequência já está pronta e, inclusive, foi escrita de maneira a ter uma definição bem clara e expressiva! Devemos ter uma quantidade de rodadas que é menor que a sequência, afinal se temos 4 luzes temos uma quantidade de rodadas que aumenta até chegar a 4. Isso significa que precisamos saber qual é a rodada em questão.

Dessa maneira, acrescentamos ao diagrama, embaixo do SETUP, um comentário mencionando que essa rodada é igual a 0. No PRONTO\_PARA\_PROX\_RODADA acrescentamos o comentário rodada++ e substituimos o Número de Luzes Respondidas por Rodada < Tamanho Sequencia. Agora, podemos retornar ao código fonte e aprender como a variável rodada funciona e se relaciona.

O primeiro passo é criar uma varíavel e como ela é importantíssima no jogo todo, então, a colocaremos junto ao escopo global. Isto é, junto ao int sequenciaLuzes(TAMANHO\_SEQUENCIA). Portanto, vamos escrever int rodada = 0;:

int sequenciaLuzes(TAMANHO\_SEQUENCIA);  
  
int rodada = 0;

Ao iniciar o jogo será perguntado ao loop() qual é o estado atual, no momento nós temos que é USUARIO\_ RESPONDENDO. Nós colocaremos que se a rodada é menor que o tamanho da sequência, assim, nós preenchemos if(rodada < TAMANHO\_SEQUENCIA) e pedimos para retornar PRONTO\_PARA\_PROX\_RODADA . E, ainda, acrescentamos um elseseguido de um return. E apagamos o comentário //Computar Estado Atual:

int estadoAtual() {  
  if(rodada < TAMANHO\_SEQUENCIA) {  
    return PRONTO\_PARA\_PROX\_RODADA;  
  }else{  
    return JOGO\_FINALIZADO\_SUCESSO;  
  }  
}

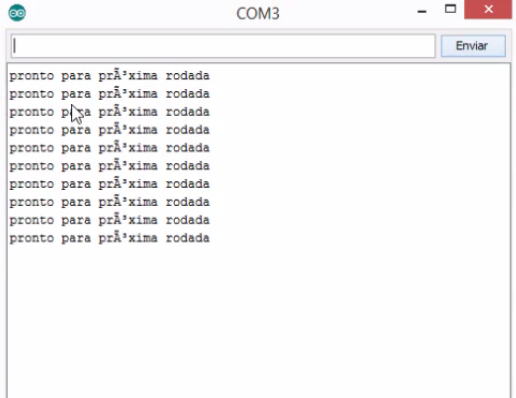
Vamos substituir o for(;;) por delay e ficaremos com o delay(500) para acompanhar melhor no serial. Podemos verificar se fica tudo certo e enviando para o **Arduino** teremos:



Ou seja, observando isso podemos concluir que ele nunca chega ao jogo finalizado com sucesso! Isso ocorre porque a rodada não está sendo incrementada, então, é preciso incrementar ela usando o rodada++ :

void loop() {  
  switch(estadoAtual()) {  
  case PRONTO\_PARA\_PROX\_RODADA;  
    Serial.println("pronto para próxima rodada");  
    rodada++;  
    break;  
  case USUARIO\_RESPONDENDO;  
    Serial.println("usuário respondendo");  
    break;  
  JOGO\_FINALIZADO\_SUCESSO;  
    Serial.println("jogo finalizado sucesso");  
    break;  
  case JOGO\_FINALIZADO\_FALHA;  
    Serial.println("jogo finalizado falha");  
    break;  
  }

E acompanhando isso no monitor serial podemos verificar o que acontece:



Agora temos algo que é mais próximo do que queríamos!

Não vamos colocar o código de próxima rodada dentro disso, pois, ele ficará sujo e difícil de ler. Por isso, criamos uma função, a preparaNovaRodada() que acrescentamos junto de:

void loop() {  
  switch(estadoAtual()) {  
  case PRONTO\_PARA\_PROX\_RODADA;  
    Serial.println("pronto para próxima rodada");  
    preparaNovaRodada();  
    rodada++;  
    break;  
  
    //...  
  
    }  
}

E, agora, podemos criar a função, void preparaNovaRodada() e dentro disso colocamos o rodada++ que nós apagamos do loop().

void loop() {  
  switch(estadoAtual()) {  
  case PRONTO\_PARA\_PROX\_RODADA;  
    Serial.println("pronto para próxima rodada");  
    preparaNovaRodada();  
    break;  
  
    //...  
  
    }  
}

Por fim, adicionamos um tocaLedsRodada. Isso tudo deve ficar em baixo do delay e teremos:

void preparaNovaRodada(){  
   rodada++;  
   tocaLedsRodada();  
}

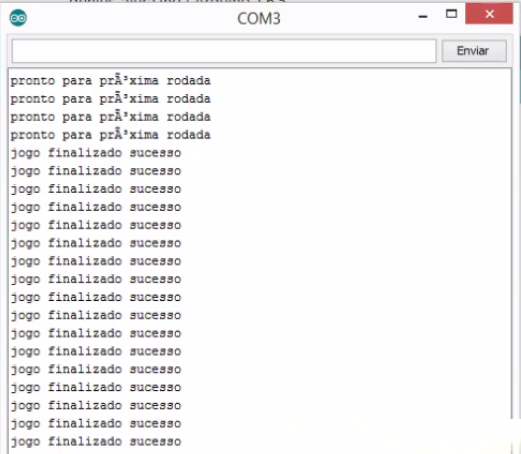
E testando isso verificamos que realmente funciona!

Toda vez que o tocaLedsRodada() for chamado, ele toca até o tamanho da sequência, mas, na verdade, queremos que o a sequência seja aumentada! Para arrumar isso basta modificar o void tocaLedsRodada para que o indice seja < rodada

void tocaLedsRodada() {  
    for(int indice = 0; rodada; indice++){  
      piscaLed(sequenciaLuzes[indice]);  
    }  
}

Podemos enviar isso para o Arduino e observar o que acontece!

Se analisarmos o que acontece é que uma cor é repetida, entretanto, nós conseguimos que a cada rodada os LEDs fossem aumentados! A mesma cor está sempre sendo reproduzida, mas agora conseguimos perceber que a sequência está sendo aumentada a cada rodada. Isso é fácil de visualizar através do *console*!



Vamos criar uma nova função para atender o estado PRONTO\_PARA\_PROXIMA\_RODADA. Nessa nova função, vamos incrementar uma variável que serve para contar a quantidade de rodadas e toca os LEDs da rodada.

1) Dessa maneira, no início do seu programa, crie uma nova variável chamada rodada:

int rodada = 0;

2) Agora, crie um função preparaNovaRodada:

void preparaNovaRodada() {  
  rodada++;  
  tocaLedsRodada();  
}

3) Implemente a função tocaLedsRodada():

void tocaLedsRodada() {  
  for (int i = 0; i < rodada; i++) {  
    piscaLed(sequenciaLuzes[i]);  
  }  
}

4) Agora, chame a função preparaNovaRodada dentro do switch no case que atende PRONTO\_PARA\_PROXIMA\_RODADA. Na função loop altere:

void loop() {  
  switch (estadoAtual()) {  
    case PRONTO\_PARA\_PROXIMA\_RODADA:  
      Serial.println("Pronto para a proxima rodada");  
      preparaNovaRodada(); //aqui  
      break;  
    case USUARIO\_RESPONDENDO:  
      Serial.println("Usuario respondendo");  
      break;  
    case JOGO\_FINALIZADO\_SUCESSO:  
      Serial.println("Jogo finalizado com sucesso");  
      break;  
    case JOGO\_FINALIZADO\_FALHA:  
      Serial.println("Jogo finalizado com falha");  
      break;  
  }  
  delay(MEIO\_SEGUNDO);  
  
}

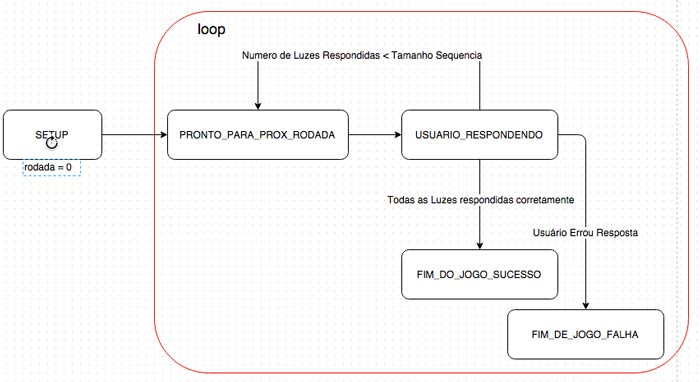
5) Por fim, altere a função estadoAtual para verificar se ainda há rodadas:

int estadoAtual() {  
  if (rodada < TAMANHO\_SEQUENCIA) {  
      return PRONTO\_PARA\_PROXIMA\_RODADA;   
  } else {  
    return JOGO\_FINALIZADO\_SUCESSO;  
  }   
  return JOGO\_FINALIZADO\_FALHA;  
}

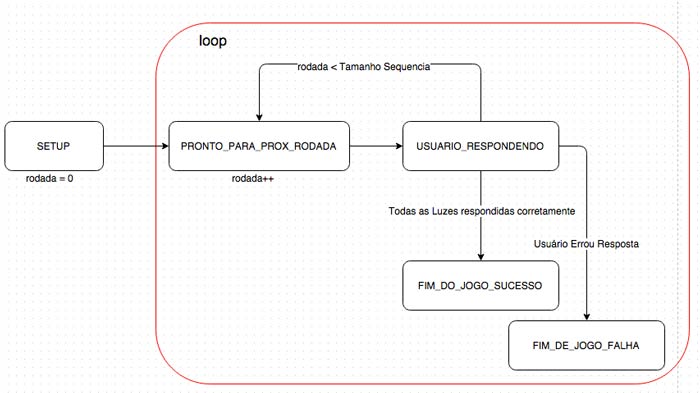
Compile e teste o seu código! Ao rodar, fique de olho no monitor serial.

Novamente, vamos pensar no diagrama de estados antes de mexer no código. Para fazer a transição entre os estados do jogo é preciso saber quantos rodadas o jogador já fez!

Por exemplo, se o jogador já fez 4 rodadas com sucesso o jogo foi finalizado com triunfo! Para saber disso, é preciso criar uma nova variável rodada que inicializaremos na função setup:



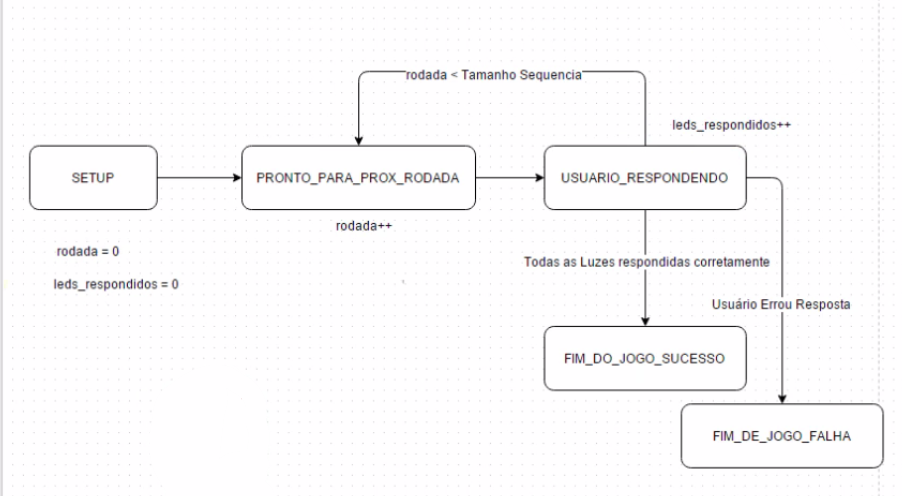
Essa variável rodada será incrementada a cada rodada e também servirá para descobrir se há uma próxima rodada:



Faça as alterações no diagrama. Você pode baixar arquivo do draw.io [aqui](https://s3.amazonaws.com/caelum-online-public/arduino/07/genius-estados-drawio-cap7-exercicio5.xml). Basta baixar o XML e abrir no na aplicação.

A medida que as rodadas vão passando, mais luzes vão acendendo até que depois da última rodada o jogo é finalizado e o jogador acaba sem nada para fazer. Isso ocorre pois não marcamos a transição entre PRONTO\_PARA\_PROX\_RODADA para USUARIO\_RESPONDENDO. E, assim como foi necessário criar uma variável para guardar a rodada, também será preciso criar uma que guarde o que o usuário respondeu.

Mas, antes, vamos pensar nisso utilizando o diagrama! E, embaixo do SET UP, adicionamos um comentário: leds\_respondidos = 0. Quando fazemos o SET UP do jogo estamos na verdade inicializando a variável, assim, toda vez que o usuário estiver respondendo vamos poder incrementar isso e fazer algo semelhante ao que temos no PRONTO\_PARA\_PROX\_RODADA. Vamos dizer que quando o estado for USUARIO\_RESPONDENDO, então, teremos leds\_respondidos++. Teremos o seguinte:



Dessa maneira, no SETUP os leds\_respondidos são iniciados, no PRONTO\_PARA\_PROX\_RODADA a rodada é incrementada e no USUARIO\_RESPONDENDO ela é incrementada a medida que os LEDs forem respondidos. Isso significa que na volta para o PRONTO\_PARA\_PROX\_RODADA temos que pensar um pouco. Se sempre só ficarmos incrementando os LEDs isso significa que nesse trecho do USUARIO\_RESPONDENDO para o PRONTO\_PARA\_PROX\_RODADA sempre ficaremos fazendo uma conta.

Facilita nossa vida saber que quando uma nova rodada é iniciada a variável é zerada, assim, quando os LEDs forem respondidos a variável leds\_respondidos será igual a rodada. Assim, se a variável leds\_respondido for igual a zero, isso significa que nada terá sido respondido! Portanto, para encerrar essa etapa é preciso responder a um LED. Ou seja, acrescentamos a condição leds respondidos == rodada para transitar de USUARIO\_RESPONDENDO para o PRONTO\_PARA\_PROX\_RODADA . Apenas quando essa condição for verdadeira é que se passará para a próxima rodada.

**Resumindo**: Apenas quando as condições rodada < Tamanho Sequencia e leds\_respondidos == rodada forem verdadeiras é que teremos uma próxima rodada!

Lembrando que é sempre bom pensar um pouco sobre o funcionamento do projeto antes de começar a executá-lo!

Após passar por toda essa reflexão já temos como controlar quantas repostas o usuário deu e o seu relacionamento com as rodadas. Vamos voltar ao código, nele implementaremos o int leds\_respondidos= 0 abaixo do int rodada = 0;:

int rodada = 0;  
int leds\_respondidos= 0

E no preparaNovaRodada() nós zeramos isso acrescentando um leds\_respondidos = 0. Teremos:

void  preparaNovaRodada() {  
  rodada++;  
  leds\_respondidos = 0;  
  tocaLedRodada();  
}

No loop() nós colocaremos o processamento da resposta, escrevemos junto ao USUARIO\_RESPONDENDO o processaRespostaUsuario. Teremos:

void loop() {  
  switch(estadoAtual()) {  
  case PRONTO\_PARA\_PROX\_RODADA;  
    Serial.print1n("pronto para próxima rodada");  
    preparaNovaRodada();  
    break;  
  case USUARIO\_RESPONDENDO;  
    Serial.print1n("usuário respondendo");  
    processaRespostaUsuario();  
    break;  
  case JOGO\_FINALIZADO\_SUCESSO;  
    Serial.print1n("jogo finalizado sucesso");  
    break;  
  case JOGO\_FINALIZADO\_FALHA;  
    Serial.print1n("jogo finalizado falha");  
    break;  
  }  
delay(500);

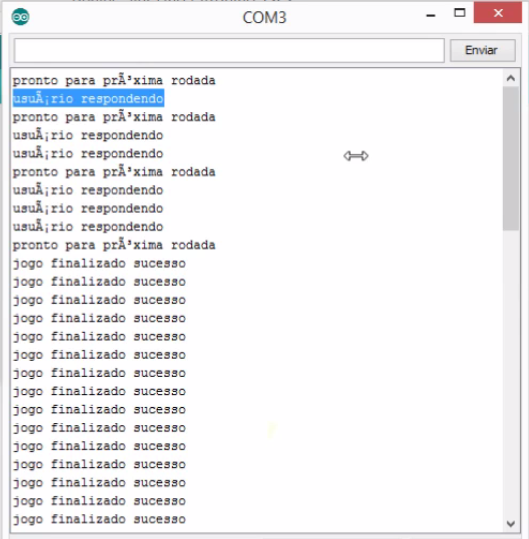
Isso requer que nós tenhamos uma função processaRespostaUsuario() que nós criaremos logo abaixo do preparaNovaRodada(). Junto a isso colocaremos o leds\_respondidos++:

void processaRespostaUsuario(){  
    leds\_respondidos++  
}

Ainda falta dizer para o programa ir ao estado USUARIO\_RESPONDENDO. Assim, temos que dizer para estadoAtual() qual é a condição necessária para isso acontecer. Nós sabemos, observando o estadoAtual, que quando a rodada for menor que o tamanho da sequência nós estamos passando direto. Então, temos que saber quando está tudo pronto para uma próxima rodada. A criação da variável if é justamente para essa situação, assim, if(leds\_respondidos == rodada). Com isso, nós estamos falando que a partir dessa condição, nós estamos prontos para uma próxima rodada, caso contrário, o usuário ainda está respondendo, para transmitir essa mensagem nós escrevemos elsee return USUARIO\_RESPONDENDO:

int estadoAtual() {  
  if(rodada < TAMANHO\_SEQUENCIA){     
     if(leds\_respondidos == rodada) {  
       return PRONTO\_PARA\_PROX\_RODADA;  
     }else{  
       return USUARIO\_RESPONDENDO;  
    }  
  }else{  
    return JOGO\_FINALIZADO\_SUCESSO;  
  }  
}

Observe como foi interessante termos feito o planejamento antes! Caso contrário teria sido tortuoso buscar respostas através da prática! Podemos compilar isso e enviar! Observe o que temos:

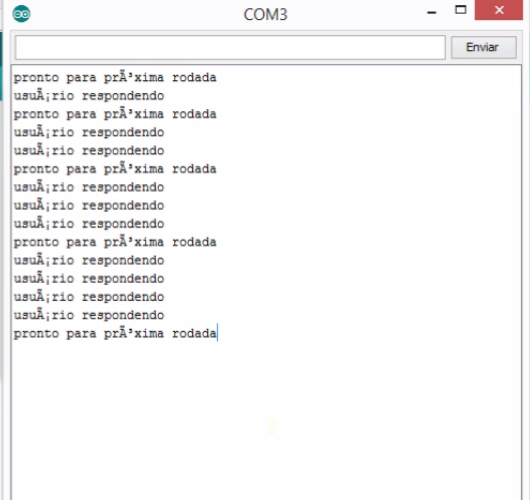


Repare que era para ter ocorrido quatro respostas, entretanto, obtivemos apenas três! Observe que o usuário não participa da próxima rodada! Na primeira rodada ele responde uma vez, na segunda, duas vezes, na terceira, três vezes, mas na quarta rodada ele não responde e é direcionado para JOGO\_FINALIZADO\_SUCESSO.

Vamos reparar no código! Nós estamos barrando o código quando dizemos que no estadoAtual() a condição é if(rodada < TAMANHO\_SEQUENCIA). Para resolver essa questão é preciso dar mais uma chance, ou seja, vamos colocar como quesito <=. Ficaremos com:

int estadoAtual() {  
  if(rodada <= TAMANHO\_SEQUENCIA){  
     if(leds\_respondidos == rodada) {  
       return PRONTO\_PARA\_PROX\_RODADA;  
     }else{  
       return USUARIO\_RESPONDENDO;  
    }  
  }else{  
    return JOGO\_FINALIZADO\_SUCESSO;  
  }  
}

Podemos mais uma vez compilar isso e enviar! Vamos observar o *console* para ver se todos os passos que desejamos são seguidos. Teremos:

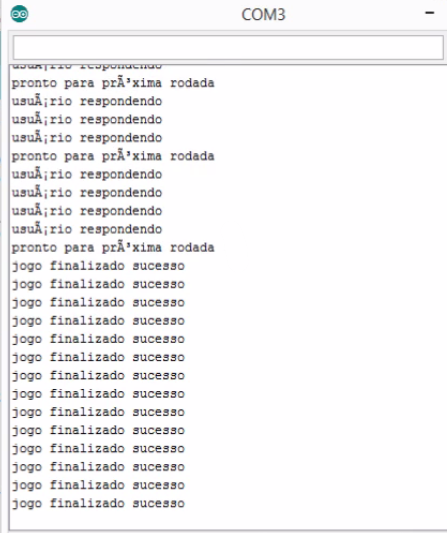


Nós ainda não conseguimos chegar ao resultado desejado, ou seja, alcançar o jogo finalizado. Isso provavelmente está relacionado a tentar tocar mais luzes do que cabem na sequência. Vamos observar o tocaLedsRodada. O indice que tínhamos descrito é aumentado em mais 1! Antes ele parava em 4 e agora ele para em 5 e, portanto, a sequência de luzes recebe um número maior que a quantidade de luzes que estão dentro da array e isso causa um erro!

Então, não podemos tocar a todo momento a sequência, o que podemos fazer para resolver esse problema é... no preparaNovaRodada acrescentar if(rodada < TAMANHO\_SEQUENCIA), assim, se o tamanho for menor que o da sequência, o LED da rodada é tocado. Teremos:

void preparaNovaRodada(){  
   rodada++;  
   leds\_respondidos = 0;  
   if(rodada < TAMANHO\_SEQUENCIA){  
     tocaLedsRoadada();  
   }  
}

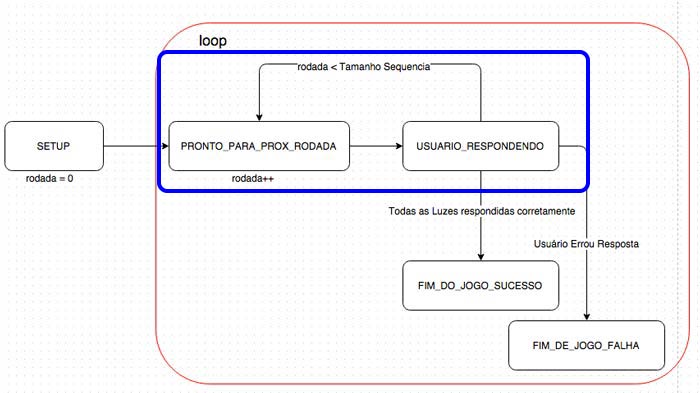
Dessa forma, os Leds serão tocados apenas se necessário. Podemos mais uma vez compilar. Observe o *console*:



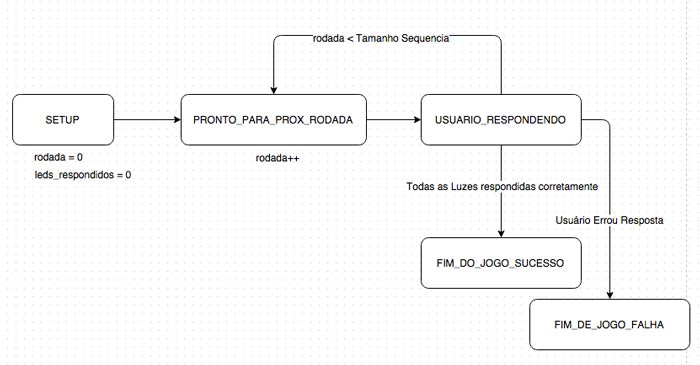
Finalmente teremos o que queríamos!

Caro aluno, se você estiver iniciando desse ponto, nós disponibilizamos o arquivo draw.io para [download](https://s3.amazonaws.com/caelum-online-public/arduino/08/genius-estados-drawio-cap8-exercicio1.xml). Você deve abrí-lo no [http:/draw.io](http://draw.io/) e fazer as seguintes alterações:

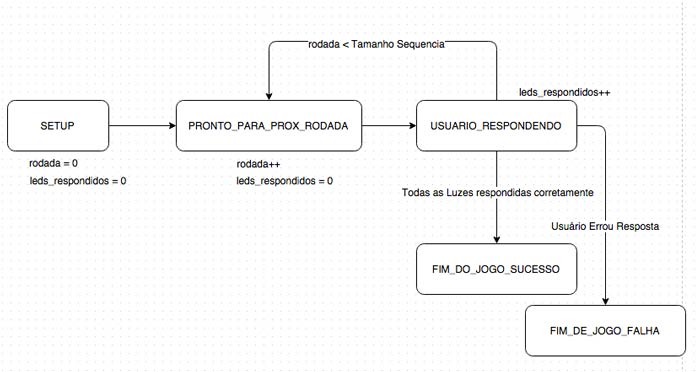
1) No último capítulo desenhamos os estados e as principais transições, no entanto, faltou ainda uma transição importante: de PRONTO\_PARA\_PROXIMA\_RODADA para USUARIO\_RESPONDENDO.



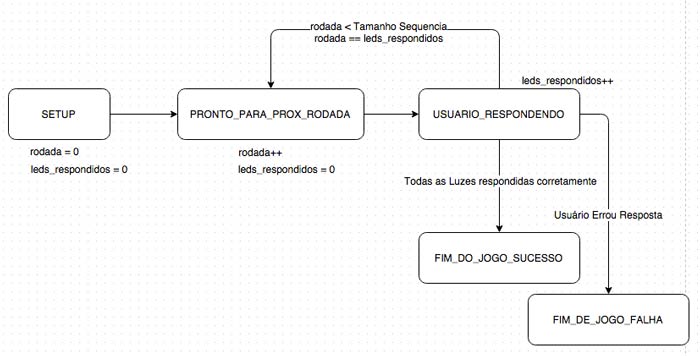
Para tanto, é preciso criar uma segunda variável que guarde quantas vezes o usuário respondeu a essa rodada. Vamos chamar essa variável de leds\_respondidos e inicializar na função setup:



2) Essa variável é incrementada cada vez que o usuário responde, mas também deve ser reinicializada a cada rodada (pois no nosso jogo, o usuário precisa começar no início da sequência luminosa):



3) Por fim, devemos deixar preparada a próxima rodada. Se a valor da rodada é igual ao LED respondido, ou seja, se na primeira rodada o jogador responder o LED corretamente, ai vem a próxima rodada. E, na segunda rodada, o jogador dever responder duas vezes a sequência com sucesso e assim por diante:



Até o momento vimos a estrutura de controle funcionando, mas ainda não processamos a resposta do usuário! Por enquanto estivemos incrementando a variável!

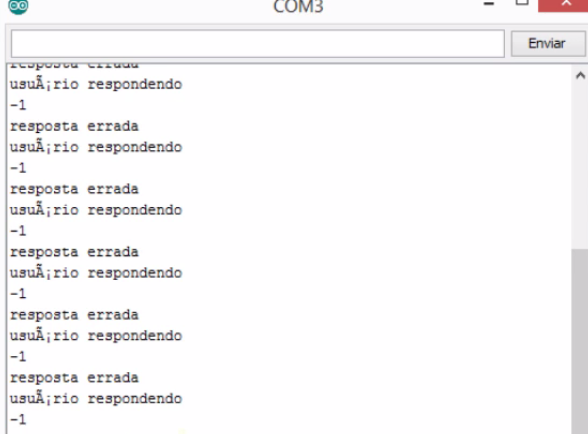
O processamento da resposta ocorre quando o usuário pressiona um botão, portanto, é preciso ler o que foi clicado como certo ou errado. Na verdade, já existe uma função que faz isso o checaRespostaJogador() e o gabarito do jogo encontra-se na sequenciaLuzes(). Em processaRespostaUsuario nós podemos adicionar o int resposta = checaRespostaJogador e perguntar se o valor contestado é correto, if(resposta == sequenciaLuzes()). Ainda, podemos acrescentar leds\_respondidos que servirá como índice dessa pergunta. Dessa maneira, ocorrerá uma comparação estabelecida pelo operador lógico == , o valor do item e o próprio elemento. Ainda, podemos adicionar um else para o caso do jogador ter contestado errado. Por fim, adicionamos o monitor serial! Teremos:

void processaRespostaUsuario() {  
  int resposta = checaRespostaJogador();  
  
  if(resposta == sequenciaLuzes[leds\_respondidos]) {  
     leds\_respondidos++;  
  }else{  
    Serial.print1n("resposta errada");    
  }  
}

Compilando isso e observando a protoboard e também o console não obtemos respostas satisfatórias sobre o que está acontecendo. Então, podemos adicionar ao processaRespostaUsuario o Serial.print1n(resposta) para melhor visualizar o que está acontecendo:

void processaRespostaJogador() {  
  int resposta = checaRespostaUsuario();  
  
  Serial.print1n(resposta);  
  
  if(resposta == sequenciaLuzes[leds\_respondidos]) {  
  }else{  
    Serial.print1n("resposta errada");    
  }  
}

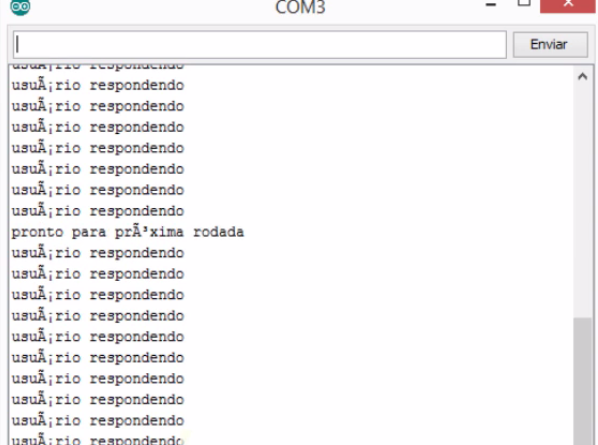
Novamente compilamos e enviamos para observar o Console e a protoboard. Com isso teremos:



Repare que a todo momento o resultado assinalado é -1. O motivo disso é porque o jogador ainda não pressionou nada! Ou seja, é preciso tratar esse momento em que o jogador ainda não escolheu nenhum botão. Assim, podemos interromper a execução e dessa maneira nós acrescentamos if(resposta == INDEFINIDO) e como essa função não retorna nada, apenas adicionamos return. Teremos o seguinte no código:

void processaRespostaUsuario() {  
  int resposta = checaRespostaJogador();  
  
  if(resposta == INDEFINIDO){  
    return;  
  }  
  
  if(resposta == sequenciaLuzes[leds\_respondidos]) {  
    leds\_respondidos++;  
  }else{  
    Serial.print1n("resposta errada");    
  }  
}

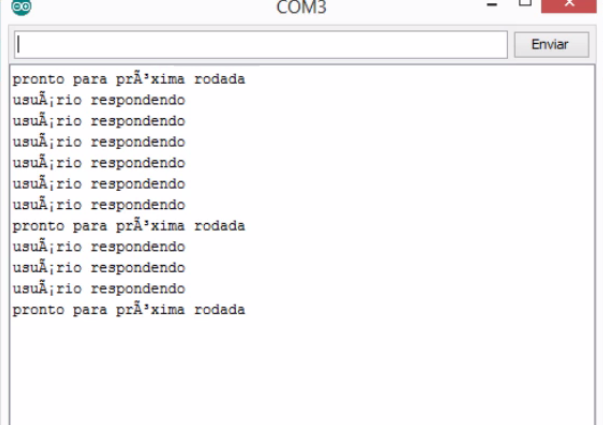
Compilando isso e mexendo no **Arduino** podemos verificar, pelo console, que enquanto aguardamos a resposta do jogador é mostrado usuário respondendo. Podemos fazer um teste e contestar errado a um LED, nesse caso, veremos que ele avisa que a resposta é errada! Assim como, ao pressionar certo ele avisa que passamos para a próxima rodada. Observe:



A partir da observação no console percebemos que ocorre um equívoco, pois em determinado ponto não é mostrado mais nada! Portanto, será preciso investigar o porquê dessa situação ocorrer! Por qual motivo não conseguimos que ele toque uma única vez! Se observarmos descobrimos que ele nunca toca quando a rodada é exatamente igual a sequência. E isso é fácil de corrigir, o preparaNovaRodada deve ser menor ou igual que TAMANHO\_SEQUENCIA. Assim, teremos o seguinte, alterando o caracter para <=:

void preparaNovaRodada(){  
  rodada++;  
  leds\_respondidos = 0;  
  if(rodada <= TAMANHO\_SEQUENCIA){  
    tocaLedsRodada();  
  }      
}

Vamos compilar e testar para verificar se funcionou! A medida que formos respondendo aos LEDS podemos observar o que acontece no Console:



Entretanto, o feedback ainda não está satisfatório. Quando o jogo for finalizado com sucesso, seria interessante se ele dissesse de alguma maneira que terminou! E se utilizarmos uma sequência que pisca os LEDs de uma única vez para sinalizar que o jogo finalizou com sucesso?

Vamos em void loop e acrescentamos jogoFinalizadoSucesso e teremos o seguinte:

void loop() {  
  switch(estadoAtual()) {  
  case PRONTO\_PARA\_PROX\_RODADA;  
    Serial.print1n("pronto para próxima rodada");  
    preparaNovaRodada();  
    break;  
  case USUARIO\_RESPONDENDO;  
    Serial.print1n("usuário respondendo");  
    processaRespostaUsuario();  
    break;  
  case JOGO\_FINALIZADO\_SUCESSO;  
    Serial.print1n("jogo finalizado sucesso");  
    jogoFinalizadoSucesso();  
    break;  
  case JOGO\_FINALIZADO\_FALHA;  
    Serial.print1n("jogo finalizado falha");  
    break;  
  }  
delay(500);

E nós renomeamos a função void piscaSequencia1(). Nós a chamaremos de jogoFinalizadoSucesso(). Teremos o seguinte:

void jogoFinalizadoSucesso(){  
  piscaLed(LED\_VERDE);  
  piscaLed(LED\_AMARELO);  
  piscaLed(LED\_VERMELHO);  
  piscaLed(LED\_AZUL);  
  delay(MEIO\_SEGUNDO);  
}

Agora temos um feedback introduzido, mas vamos aproveitar para adicionar uma sequência que só será realizada quando o usuário errar. Assim, apagamos o placaSequencia2 e trocamos por jogoFinalizadoFalha. Teremos:

void jogoFinalizadoFalha() {  
   digitalWrite(LED\_VERDE,HIGH);  
   digitalWrite(LED\_AMARELO,HIGH);  
   digitalWrite(LED\_VERMELHO,HIGH);  
   digitalWrite(LED\_AZUL,HIGH);  
   delay(UM\_SEGUNDO);  
   digitalWrite(LED\_VERDE, LOW);  
   digitalWrite(LED\_AMARELO, LOW);  
   digitalWrite(LED\_VERMELHO, LOW);  
   digitalWrite(LED\_AZUL, LOW);  
   delay(MEIO\_SEGUNDO);  
 }

E adicionamos o jogoFinalizadoFalha junto ao void loop() :

void loop() {  
  switch(estadoAtual()) {  
  case PRONTO\_PARA\_PROX\_RODADA;  
    Serial.print1n("pronto para próxima rodada");  
    preparaNovaRodada();  
    break;  
  case USUARIO\_RESPONDENDO;  
    Serial.print1n("usuário respondendo");  
    processaRespostaUsuario();  
    break;  
  case JOGO\_FINALIZADO\_SUCESSO;  
    Serial.print1n("jogo finalizado sucesso");  
    jogoFinalizadoSucesso();  
    break;  
  case JOGO\_FINALIZADO\_FALHA;  
    Serial.print1n("jogo finalizado falha");  
    jogoFinalizadoFalha();  
    break;  
  }  
delay(500);

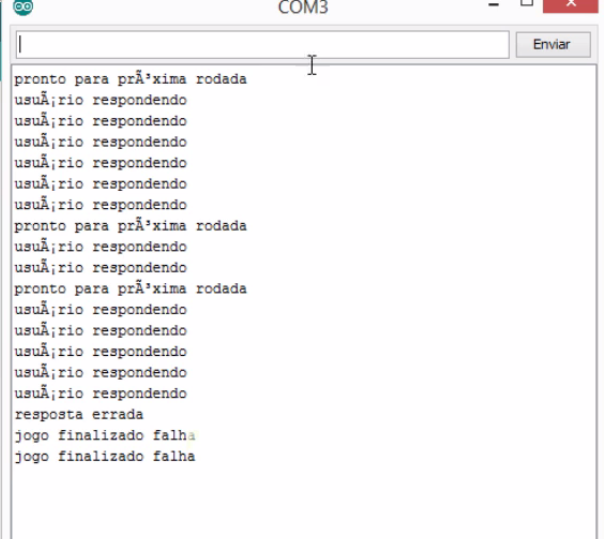
Agora os *feedbacks* estão prontos. Mas, ainda tem uma questão a ser resolvida! Vamos observar o estadoAtual! Como sabemos que estamos no estado do jogo com falha ou sucesso? Nós adicionamos ao else o if e rodada == TAMANHO\_SEQUENCIA +1. Ainda, acrescentamos outro else e colocamos return JOGO\_FINALIZADO\_FALHA. Teremos:

int estadoAtual() {  
  if(rodada <= TAMANHO\_SEQUENCIA)  
}  
    if(leds\_respondidos == rodada){  
      return PRONTO\_PARA\_PROX\_RODADA;  
    }else{  
       return USUARIO\_RESPONDENDO;  
    }  
  }else if(rodada == TAMANHO\_SEQUENCIA + 1){  
     return JOGO\_FINALIZADO\_SUCESSO;  
  }else{  
     return JOGO\_FINALIZADO\_FALHA;  
  }  
}

Mas, em nenhum momento conseguimos cair nesse caso! Estamos interagindo com um jogo de memória onde a regra do jogo é que se o usuário respondeu errado, temos que parar o jogo e avisar que ele errou. No processaRespostaUsuario falamos que a contestação é errada! Assim, acrescentamos rodada = TAMANHO\_SEQUENCIA + 2; . Observe o que acontece:

void processaRespostaUsuario() {  
  int resposta = checaRespostaJogador();  
  
   if(resposta == INDEFINIDO){  
      return;  
   }  
  
  if(resposta == sequenciaLuzes[leds\_respondidos]) {  
     leds\_respondidos++;  
  }else{  
    Serial.print1n("resposta errada");    
    rodada = TAMANHO\_SEQUENCIA + 2;  
  }  
}

Vamos observar o console! Vamos simular um jogo, nele acertamos algumas perguntas e depois erramos uma. Perceba que no caso do erro recebemos a resposta jogo finalizado falha:



Agora já temos todas as informações necessárias para criar o jogo **Genius**!!!

Chegou o momento de verificar a resposta do usuário e deixar o nosso jogo completo! Já temos uma função para isso, mas, por enquanto, ela só incrementa a variável leds\_respondidos. Procure no seu código a função processaRespostaUsuario:

void processaRespostaUsuario(){  
    leds\_respondidos++;  
}

Vamos começar?!

1) Primeiro, é preciso alterar a função e incrementar a variável leds\_respondidosapenas quando a resposta do jogador estiver correta. Lembrando que já temos uma outra função checaRespostaJogador que verifica a resposta do jogador. Chamamos essa função e adicionamos um if que testará se a resposta do usuário é equivalente a sequência de luzes proposta pelo jogo:

void processaRespostaUsuario(){  
  int resposta = checaRespostaJogador();  
  
  if(resposta == sequenciaLuzes[leds\_respondidos]){  
    leds\_respondidos++;  
  }else{  
    //aqui terá um codigo ainda  
    Serial.println("resposta errada");  
  }  
}

2) O problema ainda é que o jogador pode demorar para responder e nesse caso, a função checaRespostaJogador devolve INDEFINDO. Então, é preciso verificar se recebemos o INDEFINIDO . Se isso acontecer, devemos sair da função já que não há resposta do jogador:

void processaRespostaUsuario(){  
  int resposta = checaRespostaJogador();  
  
  if(resposta == INDEFINIDO){ //novo if  
    return; //saindo do função  
  }  
  
  if(resposta == sequenciaLuzes[leds\_respondidos]){  
    leds\_respondidos++;  
  }else{  
    Serial.println("resposta errada");  
  }  
}

3) Falta retornar ao jogador um feedback sobre seus erros ou acertos em relação a sequência de luzes. Portanto, vamos aproveitar as funções piscaSequencia1 e piscaSequencia2. Renomeie as funções para:

piscaSequencia1 para jogoFinalizadoSucesso piscaSequencia2 para jogoFinalizadoFalha

4) Agora, procure a função loop. Nela, dentro do case do estado JOGO\_FINALIZADO\_SUCESSO, chame a função jogoFinalizadoSucesso. Repita o mesmo procedimentos para o case do estado JOGO\_FINLIZADO\_FALHA chamando a função jogoFinalizadoFalha:

switch(estadoAtual()){  
    case PRONTO\_PARA\_PROX\_RODADA:  
      Serial.println("pronto para próxima rodada");  
      preparaNovaRodada();  
      break;  
    case USUARIO\_RESPONDENDO:  
      Serial.println("usuário respondendo");  
      processaRespostaUsuario();  
      break;  
    case JOGO\_FINALIZADO\_SUCESSO:  
      Serial.println("jogo finalizado sucesso");  
      jogoFinalizadoSucesso(); //novo  
      break;  
    case JOGO\_FINLIZADO\_FALHA:  
      Serial.println("jogo finalizado falha");  
      jogoFinalizadoFalha(); //novo  
      break;  
  }  
  delay(500);  
}

5) Falta atender ainda o estado JOGO\_FINLIZADO\_FALHA na função estadoAtual. Procure a função estadoAtual e altere-a da seguinte maneira:

int estadoAtual(){  
  if(rodada <= TAMANHO\_SEQUENCIA){  
    if(leds\_respondidos == rodada){  
      return PRONTO\_PARA\_PROX\_RODADA;  
    }else{  
      return USUARIO\_RESPONDENDO;  
    }  
  }else if(rodada == TAMANHO\_SEQUENCIA + 1){ //novo if else  
    return JOGO\_FINALIZADO\_SUCESSO;  
  }else{ //novo else  
    return JOGO\_FINLIZADO\_FALHA;  
  }  
}

6) Por último, é preciso modificar a função processaRespostaUsuario, pois estamos imprimindo uma mensagem somente quando o usuário responde errado. A versão final (!) da função processaRespostaUsuario fica da seguinte forma:

void processaRespostaUsuario(){  
  int resposta = checaRespostaJogador();  
  
  if(resposta == INDEFINIDO){  
    return;  
  }  
  
  if(resposta == sequenciaLuzes[leds\_respondidos]){  
    leds\_respondidos++;  
  }else{  
    rodada = TAMANHO\_SEQUENCIA + 2; //novo  
  }  
}

Compile, teste e divirta-se!

Vamos implementar o nosso planejamento e alterar o código! Siga os passos:

1) No **ArduinoIDE**, abra o código do seu programa Genius.

2) Crie uma nova variável leds\_respondidos (ela deve estar localizada abaixo da variável rodada, por exemplo):

int leds\_respondidos = 0;

3) Essa variável será inicializada a cada rodada. Altere a função preparaNovaRodada:

void preparaNovaRodada() {  
  rodada++;  
  leds\_respondidos = 0; //novo  
  tocaLedsRodada();  
}

4) Agora, nós vamos trabalhar no processamento da resposta do jogador. Procure a função loop e dentro dela o case que atende o estado USUARIO\_RESPONDENDO. Nesse case chamaremos uma nova função, a processaRespostaUsuario:

void loop() {  
  switch (estadoAtual()) {  
    case PRONTO\_PARA\_PROXIMA\_RODADA:  
      Serial.println("Pronto para a proxima rodada");  
      preparaNovaRodada();  
      break;  
    case USUARIO\_RESPONDENDO:  
      Serial.println("Usuario respondendo");  
      processaRespostaUsuario(); //novo  
      break;  
    case JOGO\_FINALIZADO\_SUCESSO:  
      Serial.println("Jogo finalizado com sucesso");  
      break;  
    case JOGO\_FINALIZADO\_FALHA:  
      Serial.println("Jogo finalizado com falha");  
      break;  
  }  
  delay(MEIO\_SEGUNDO);  
  
}

5) Vamos criar essa função processaRespostaUsuario. Abaixo da função loop adicione o seguinte:

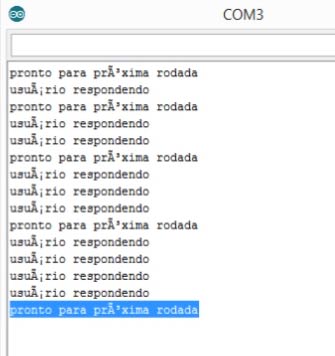
void processaRespostaUsuario(){   
    leds\_respondidos++;  
}

5) Por fim, altere a função estadoAtual para verificar se o jogador ainda precisa responder:

int estadoAtual() {  
  if (rodada <= TAMANHO\_SEQUENCIA) {  
  
    //novo if-else  
    if(leds\_respondidos == rodada){  
      return PRONTO\_PARA\_PROX\_RODADA;  
    }else{  
      return USUARIO\_RESPONDENDO;  
    }  
  
  } else {  
    return JOGO\_FINALIZADO\_SUCESSO;  
  }   
}

Compile e teste o seu código! Ao rodar, fique de olho no monitor serial.

Ao realizar o teste verificamos que, ao final do jogo, não aparece a mensagem *jogo finalizado com sucesso*. Observe:



O problema ainda é que a variável rodada será maior que o tamanho da sequência. Por isso, devemos proteger a função preparaNovaRodada e só tocar os LEDs se a rodada for *menor ou igual do que sequência*. Assim, é preciso alterar a função preparaNovaRodada:

void preparaNovaRodada(){  
  rodada++;  
  leds\_respondidos = 0;  
  if(rodada <= TAMANHO\_SEQUENCIA){ //novo if  
    tocaLedsRodada();  
  }  
}

Até agora nós vimos a estrutura de controle funcionando, mas ainda não processamos a resposta! Vamos fazer isso no próximo exercício?

Sabemos que a partida terminou quando o número de rodadas é superior a TAMANHO\_SEQUENCIA. Como isso acontece?

Quando o jogador termina a última rodada, a variável rodada será igual a QUANTIDADE\_SEQUENCIA. Perceba que o valor ainda não é maior, logo, para o código ainda estaremos no estado PRONTO\_PARA\_PROX\_RODADA. Mais uma rodada (a quinta) será executada e a variável rodada incrementada, porém com uma ligeira mudança. Nesse caso, nenhum LED piscará, porque o número da rodada é superior a TAMANHO\_SEQUENCIA, faz sentido, já que não temos um quinto LED.

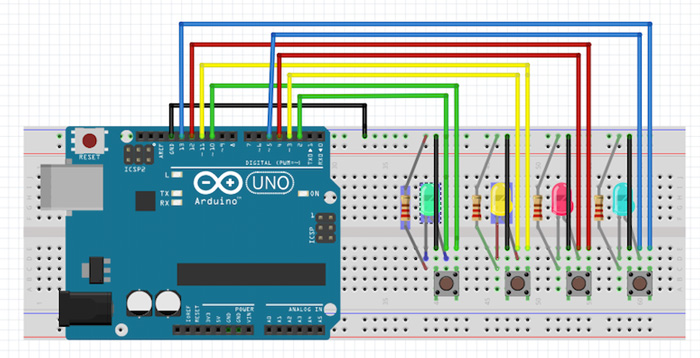
Quando o código verifica o estado em que nos encontramos ele sabe que é o FIM do jogo porque rodadas é maior que TAMANHO\_SEQUENCIA, mas isso não é o suficiente. Como ele saberá se é um sucesso ou um fracasso?

No caso do sucesso, basta testar se rodadas é igual a TAMANHO\_SEQUENCIA + 1. Se for, é simples, o jogo terminou com sucesso! Agora, se o resultado for qualquer número diferente, é porque terminou com fracasso. Quando o jogador erra, atribuímos a variável rodada o valor de TAMANHO\_SEQUENCIA + 2. Isso faz com que nossa rodada seja 6. Quando nosso programa for processar o estado, ele verifica que é o fim da partida e detecta que rodadas não é igual a TAMANHO\_SEQUENCIA + 1, caindo, assim, na condição else de fracasso.

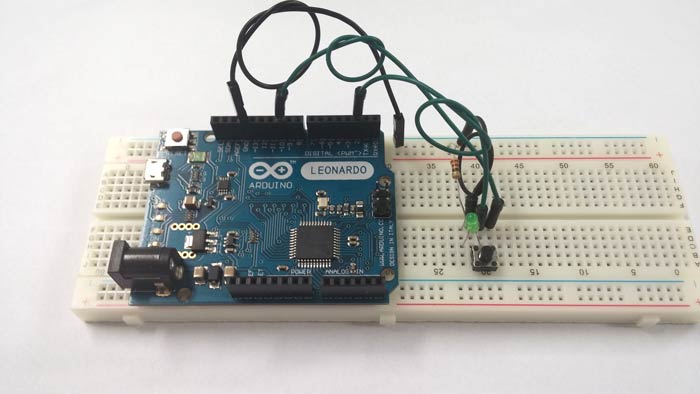
Agora que finalizamos o jogo, podemos começar a fazer pequenas melhorias nele.

Um dos aspectos que nos auxiliou muito, foi o ponto de vista didático, isto é, o modo como organizamos os nossos cabos. Seguimos um passo a passo que mostrou de modo muito claro como funcionava o circuito eletronicamente. E, ao final de tudo, apesar de obtermos um jogo funcional, ainda ficamos com os botões um pouco mal posicionados. Portanto, corremos o risco de que um usuário um pouco mais distraído esbarre nos cabos que passam por cima do botão e desconecte-os sem querer.

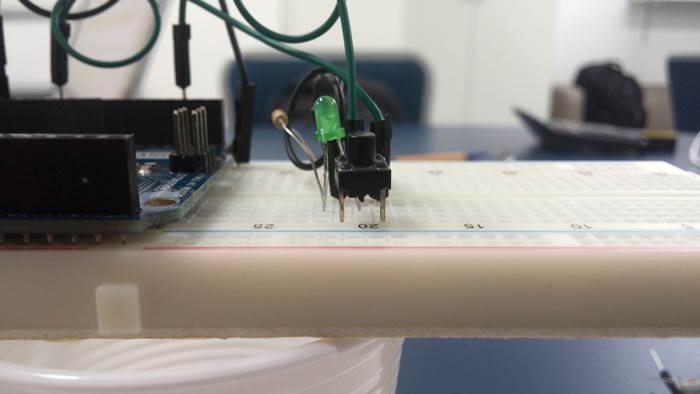
Para melhorar a experiência do usuário e resolver este problema, vamos refazer a montagem dos componentes de modo que os botões fiquem mais acessíveis.



Veja abaixo, em detalhes, como faremos isso. Exemplificaremos através da montagem de um botão.

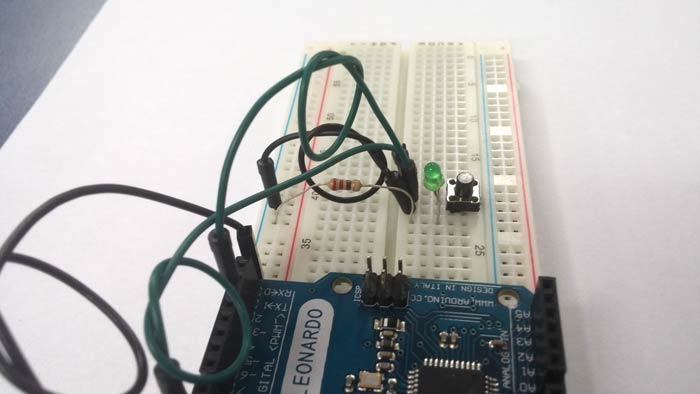


A primeira coisa que você deve ter notado é que deixamos de usar o GND (Ground) da parte de baixo do nosso Arduino, fizemos isso para isolar a parte de cabos que ligam ao terra da parte de cima da protoboard. Dessa maneira, deixamos espaço livre para o acesso ao botão embaixo. Estamos usando o GND que fica ao lado das portas digitais do Arduino e usando a linha de cima para o Ground.

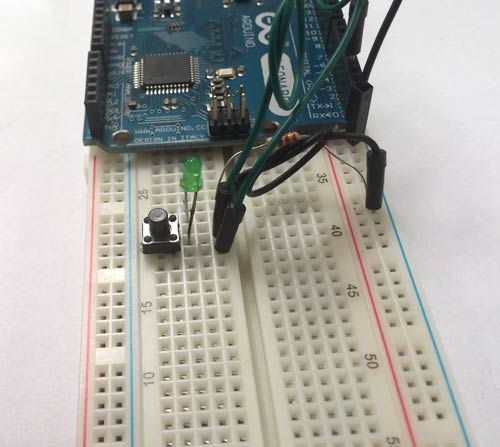




Outro detalhe importante que você deve notar, é que estamos utilizando o espaço entre os 2 terminais(as perninhas) do botão para colocar a parte positiva do LED.



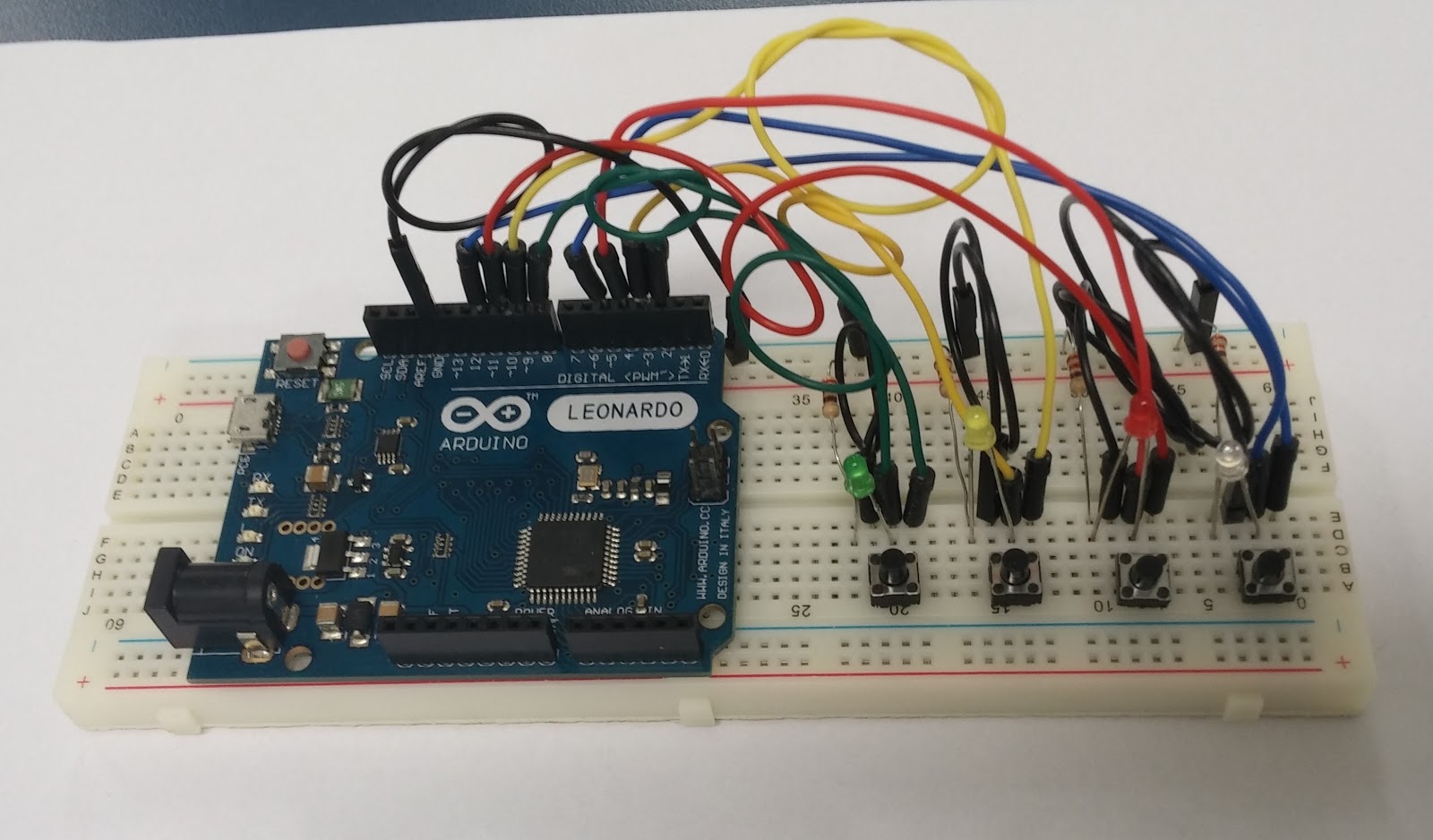
Visão lateral, utilizamos o resistor para fazer a ponte entre a parte negativa do LED e a linha do GND.



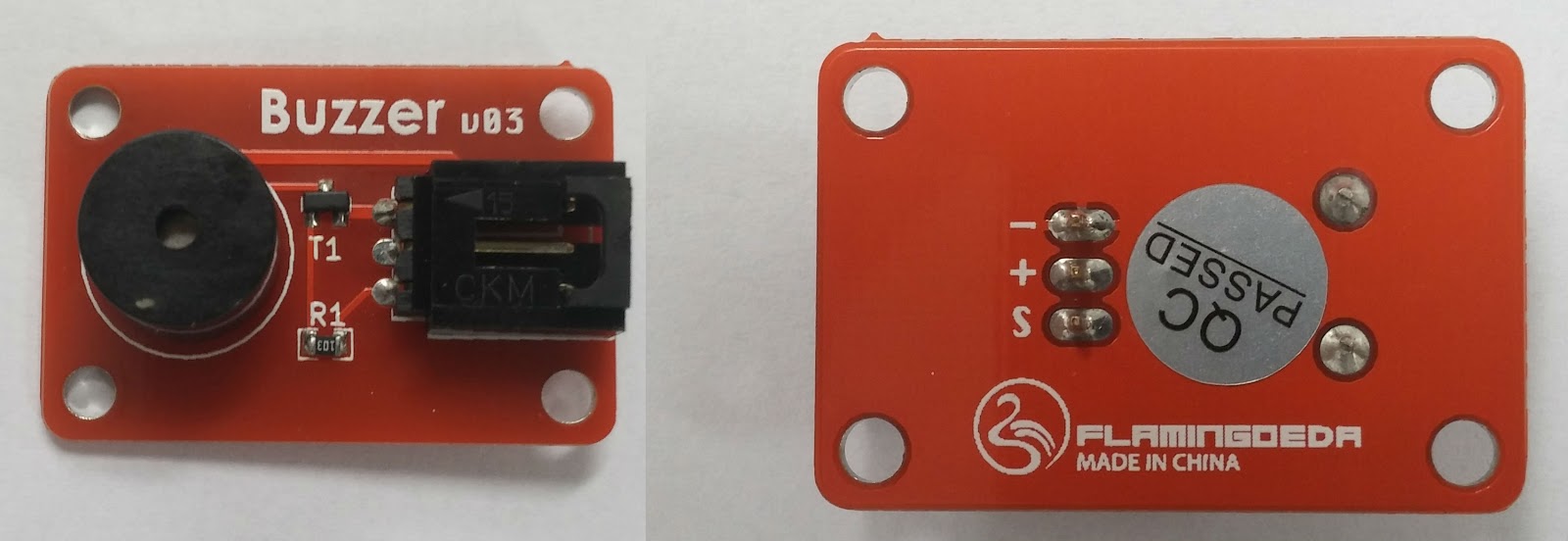
Ainda, estamos utilizando um pequeno cabo preto para fazer a conexão dos botões com a linha de GND.

Siga os passos exemplificados acima e repita o mesmo processo para todos os botões. Veja como obtemos uma visão mais limpa e adquirimos uma experiência muito melhor para os nossos jogadores.

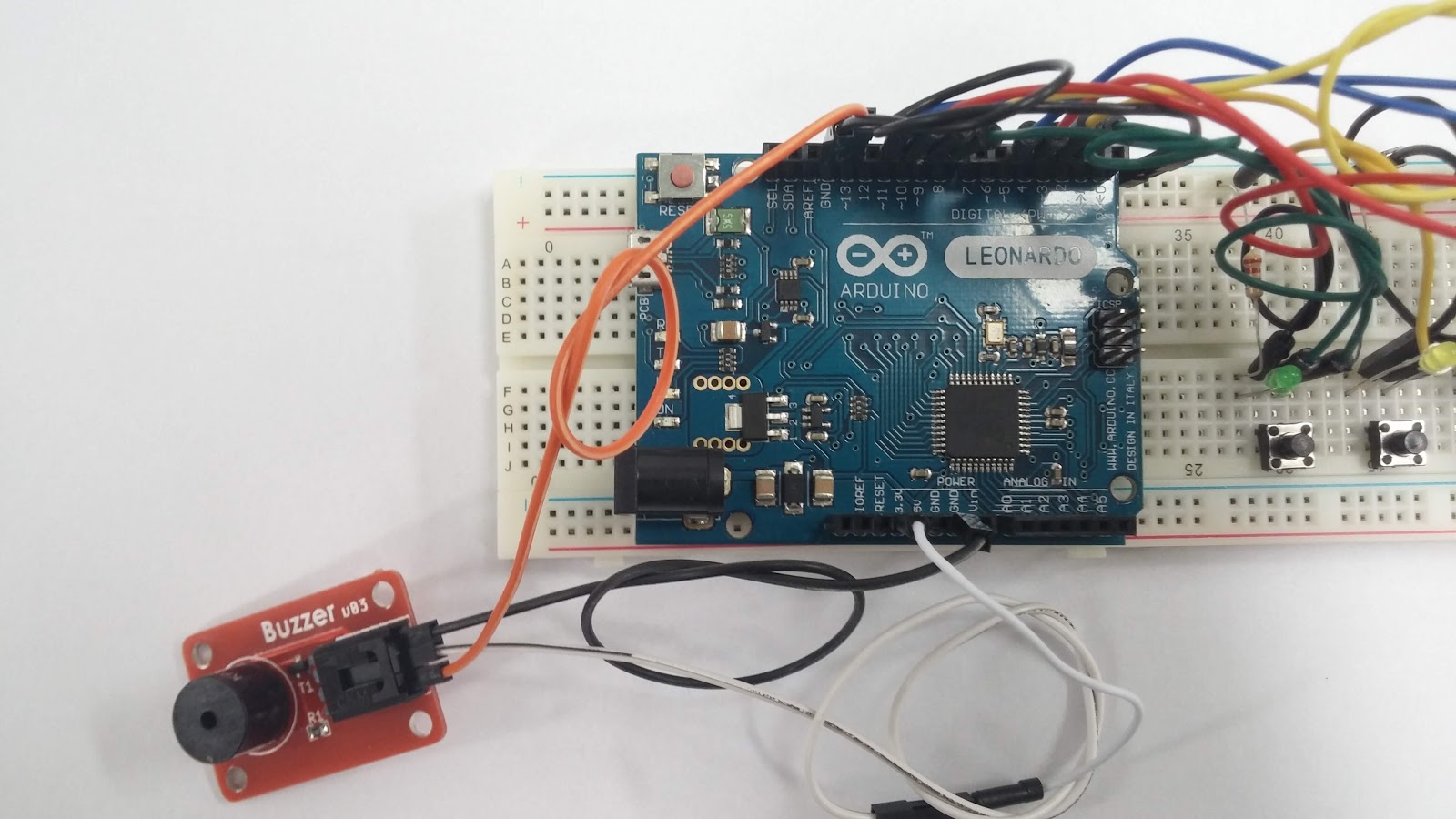
Veja como deve ficar o circuito no final:



Para esse exercício, utilizaremos um buzzer que contém um pequeno circuito integrado, como o do modelo abaixo:



Repare que nele há 3 saídas. A positiva (+) nós iremos conectar no 5 volts, a negativa (-) no terra (GND) e a S ligaremos em uma porta digital, no nosso caso usaremos a 13:



Com o buzzer já conectado ao Arduino, vamos ao código!

1) Primeiro, criamos a constante referente ao buzzer:

#define BUZZER 13

2) Segundo, usamos o 13 porque o buzzer está ligado à porta 13 do Arduino. Agora, precisamos iniciar a porta do buzzer como OUTPUT, pois vamos emitir um som. Em iniciaPortas(), adicione:

pinMode(BUZZER, OUTPUT);

3) Vamos, agora, criar uma função chamada tocaSom(). Ela recebe a frequência do som por parâmetro e utiliza essa frequência para gerar um tom no buzzer. Portanto, a função tone recebe além do buzzer e da frequência, também o tempo de duração do som por parâmetro, que nós vamos colocar como 100 milissegundos:

void tocaSom(int frequencia) {  
  tone(BUZZER, frequencia, 100);  
}

4) Com a função criada, já é possível tocar um som através de uma frequência, basta chamar a tocaSom()! Mas, quando exatamente queremos que o som seja tocado? Quando o LED piscar! Então, vamos lá na função piscaLed() e adicionamos a chamada da tocaSom():

int piscaLed(int portaLed) {  
  
  tocaSom(2000);  
  
  digitalWrite(portaLed,HIGH);  
  delay(UM\_SEGUNDO);  
  digitalWrite(portaLed,LOW);  
  delay(MEIO\_SEGUNDO);  
  
  return portaLed;  
}

5) Agora, você pode compilar o código, enviá-lo ao Arduino e testar o som. Ao fazer isso repare que sempre é tocado o mesmo som! Isso ocorre porque só reproduzimos o tom da frequência 1000. Para melhorar nosso jogo podemos tocar um som distinto para cada LED do Arduino.

Por isso, vamos criar mais uma função e a chamaremos de verificaSomDoLed(). Essa função irá receber a porta do LED por parâmetro, verificar qual LED é e dependendo de qual for, um som diferente será tocado:

void verificaSomDoLed(int portaLed) {  
  switch (portaLed) {  
    case LED\_VERDE:  
      tocaSom(2000);  
      break;  
    case LED\_AMARELO:  
      tocaSom(2200);  
      break;  
    case LED\_VERMELHO:  
      tocaSom(2400);  
      break;  
    case LED\_AZUL:  
      tocaSom(2500);  
      break;  
  }  
}

6) Repare que cada LED terá um som em uma frequência distinta. Por isso, precisamos alterar a função piscaLed() para acrescentar a verificaSomDoLed() que acabamos de cria e não mais o tocaSom():

int piscaLed(int portaLed) {  
  
  verificaSomDoLed(portaLed);  
  
  digitalWrite(portaLed,HIGH);  
  delay(calculaTempoLed());  
  digitalWrite(portaLed,LOW);  
  delay(MEIO\_SEGUNDO);  
  
  return portaLed;  
}

7) Compile e envie o código do Arduino. Simule o jogo e veja... Não ficou melhor?!

8) Para finalizar, vamos adicionar mais um som, uma espécie de sirene para servir de alerta quando o jogo for finalizado com falha. Para isso basta adicionar **no começo** da função jogoFinalizadoFalha():

void jogoFinalizadoFalha() {  
   tocaSom(300);  
   // Restante do código comentado  
}