Universidade Federal de Santa Catarina
Centro Tecnológico
Departamento de Automação e Sistemas
Engenharia de Controle e Automação
Introdução à Informática para Controle e Automação - DAS 5334

Aula 8 - Genius[1]

1 Números aleatórios

O estudo de números aleatórios é uma área fascinante da matemática e da computação. Numa série de números aleatórios, um número da série é independente dos números anteriores. Em computação, o desafio é gerar uma série numérica em que os números pareçam independentes uns dos outros. Nesse caso, diz-se que o número gerado é pseudo-aleatório. A geração de números pseudo-aleatórios depende de um "semente". Para cada semente, uma série de números pseudo-aleatórios será gerada. Não é possível gerar duas séries diferentes com a mesma semente.

Felizmente, a linguagem Arduino nos oferece uma função apropriada para gerar números aleatórios, ou melhor, números pseudo-aleatórios. A função `random()` recebe dois argumentos, o valor mínimo (incluído) e máximo (excluído) para o valor a ser gerado. Opcionalmente o valor mínimo pode ser omitido e será interpretado como sendo zero, isto é, apenas um argumento é fornecido. A função devolve um valor inteiro nos limites estipulados. Para que não seja gerada sempre a mesma série, antes do uso da função , usa-se a função `randomSeed()` que recebe como argumento a semente que queremos usar para gerar nossa série. Curioso notar que se quisermos gerar sequências pseudo-aleatórias diferentes, teremos que usar sementes diferentes. Como escolher as sementes? O ideal é que as sementes fossem escolhidas arbitrariamente ou por meio de um sorteio aleatório. Isso seria equivalente a usar `random()` para gerar uma semente para `random()`. E agora? Por coincidência, um pino da placa Arduino que não está conectado tem uma tensão "flutuante" que varia de maneira imprevisível. Assim, a função `analogRead()` pode ser usada para obter um valor a ser usado como semente. Por exemplo:

```
1 int numAleat;
2 randomSeed(analogRead(13));
3 numAleat = random(100);
4 lcd.println(numAleat);
```

O exemplo, usa uma leitura do pino `13` (desconectado) para definir uma semente. O número aleatório gerado estará no intervalo $[0, 100[$.

2 Tons

O Arduino permite gerar tons, no sentido musical, a partir de pinos PWM. No Arduino isso corresponde a gerar uma onda quadrada com determinada frequência. Quando conectado a um *piezzo buzzer* ou algum outro tipo de alto falante, um som naquela frequência será gerado. Com o Arduino Micro é possível trabalhar com frequências de 31 a 65535 Hz.

Para gerar o tom, ou a onda quadrada de uma determinada frequência, usa-se a função `tone()`. Essa função recebe dois ou três argumentos. O primeiro argumento é o pino, o segundo argumento é a frequência, e o terceiro argumento (opcional) é a duração em milisegundos. Nenhum valor é devolvido. Quando a duração não é especificada, a onda quadrada será gerada até que ocorra uma chamada à função `noTone()`, que recebe como argumento apenas a identificação do pino. As duas funções estão listadas na página de referência da linguagem Arduino na seção *Advanced I/O*.

Atenção para o fato de que uma chamada a `tone()` altera uma chamada anterior, mesmo que uma duração tenha sido especificada. Assim, pode ser necessário usar a função `delay()` com o mesmo valor usado na chamada de `tone()`. Pode também ser de interesse criar um breve intervalo, talvez imperceptível, de silêncio entre duas chamadas a `tone()`.

O exemplo a seguir mostra exemplos de chamadas a `tone()` e `noTone()`.

[1] Baseado em conteúdo do livro Think Python 2nd Edition by Allen B. Downey e conteúdo da página oficial do Arduino.

```
tone(13, 440, 500); // onda quadrada de 440 Hz no pino 13 por meio segundo
delay(500);         // meio segundo de espera

delay(10);          // pausa entre tons

tone(13, 880);      // onda quadrada de 440 Hz no pino 13
delay(500);         // meio segundo de espera
noTone(13);         // interrompe a onda quadrada
```

Notar que por limitações do *hardware*, não é possível usar o PWM do pino 5 enquanto `tone()` estiver sendo usada em algum dos outros pinos. Além disso, só é possível usar `tone()` em um pino por vez.

3 Tempo

Para resolver o problema da aula de hoje, pode ser útil, mas não é estritamente necessária, a função `millis()`. Essa função não recebe nenhum argumento e devolve o tempo em milisegundos desde que a placa foi ligada. O valor devolvido é de um tipo que ainda não vimos, `unsigned long`. O tipo `long` é um tipo de inteiro que permite uma faixa de números maior do que o que é possível com o tipo `int`. Enquanto uma variável `int` ocupa 2 bytes de memória e pode representar números de −32768 a 32767, variáveis `long` ocupam 4 bytes e podem representar valores de −2.147.483.648 a 2.147.483.647! O adjetivo `unsigned` quer dizer que se trata de um inteiro sem sinal, isto é, todos os números são positivos, permitindo números positivos ainda maiores do que com `long`. Uma variável `unsigned long` pode ter valores de 0 a 4.294.967.295. A listagem a seguir apresenta um exemplo:

```
#include <LiquidCrystal.h>
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);

unsigned long time;

void setup(){
  lcd.begin(16,2);
  randomSeed(analogRead(0));
}

void loop(){
  time = millis();
  delay(500+random(1000));
  // mostra o tempo decorrido entre as duas chamadas à milli()
  lcd.print(millis()-time);
}
```

4 *Debugging*

O circuito desta aula propositalmente não tem um *display*. O *display* é muito cômodo para a tarefa de *debugging* e esperamos que a ausência desse recurso o force a usar a ferramenta `Debugger` do Autodesk Circuits, além dos próprios LEDs do circuito. Por outro lado, e talvez até de forma contraditória às intenções de remover o *display* do circuito desta aula, o fato de estarmos usando um ambiente de simulação traz diversas facilidades. Alguns testes, ou até mesmo experimentar as funções `random()`, `randomSeed()`, e `millis()`, podem ser feitos em circuitos de aulas passadas que dispõem do *display*!

5 Problemas

Nesta aula resolveremos um único problema. É um problema um pouco maior do que os que vínhamos resolvendo, mas que com os conhecimentos adquiridos até agora poderá ser resolvido com sucesso. Esse problema porá à prova nossa habilidade de “dividir tarefas grandes e complicadas em subtarefas cada vez menores e simples, até que essas subtarefas possam ser realizadas por meio de uma das instruções disponibilizadas pela linguagem escolhida”. As tarefas para o problema que

será apresentado a seguir nem são tão grandes nem tão complicadas, mas terão que ser divididas. Além disso, será necessário “pensar criativamente na sua solução” e a linguagem Arduino permitirá “expressar essa solução com clareza e precisão”.

Acesse o circuito Genius e faça uma cópia em sua própria conta com um clique no botão `Copiar e Tinker`. Esse circuito (Figura 1) possui um *piezzo buzzer* (círculo preto), quatro LEDs, um vermelho, um amarelo, um azul e um verde, e cinco *push buttons*, cada um ao lado de um dos componentes anteriores. A Tabela 1 indica os pinos aos quais os componentes estão conectados, já com sugestão de identificadores.

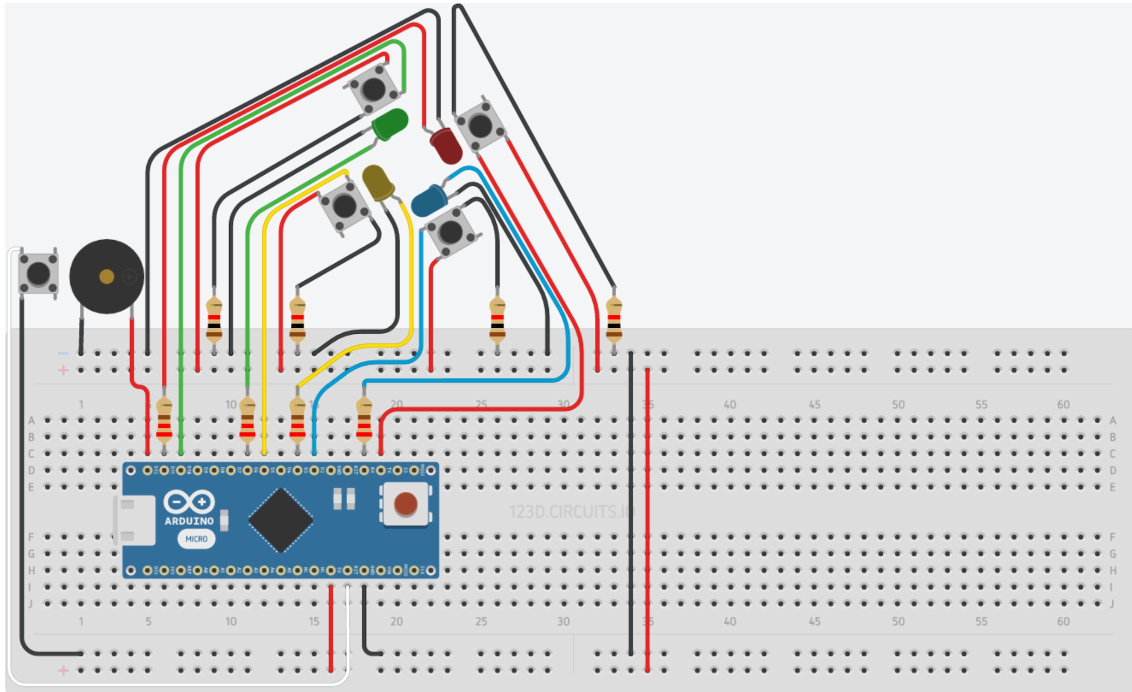

Figura 1: Circuito do Genius.

Tabela 1: Identificação dos pinos para o problema desta aula.

Componente	Pino
`redLED`	`D11`
`blueLED`	`D0`
`yellowLED`	`D3`
`greenLED`	`D6`
`redBtn`	`D1`
`blueBtn`	`D2`
`yellowBtn`	`D5`
`greenBtn`	`D10`
`buzzer`	`D12`
`resetBtn`	`RESET`

Genius (veja também Genius em anos80.net) foi o primeiro jogo eletrônico comercializado no Brasil, um sucesso na década de 1980. Nos EUA era chamado de Simon. Além de algumas chaves seletoras e botões, o dispositivo possui quatro botões luminosos nas cores verde, vermelha, azul e amarela cada um com um tom diferente associado. O Genius permite mais de um modo de jogo, mas o principal e mais conhecido consiste da geração de sequências cor/tons que devem ser repetidas pelo jogador. O jogo começa com um par cor/tom e a cada repetição bem sucedida pelo jogador, o jogo acrescenta um novo par cor/tom. O jogador repete a sequência apertando botão da respectiva cor que acende e emite o som respectivo. Se o jogador errar, o botão errado que foi pressionado acende e um tom de erro é emitido.

No circuito da Figura 1 cada LED e o botão próximo à ele correspondem a um botão do jogo Genius. O *piezzo buzzer* será usado para gerar os tons correspondentes a cada cor conforme a Tabela 2. O botão ao lado do *piezzo buzzer* é o botão de *reset* e permite reiniciar o jogo.

Tabela 2: Tons e cores do jogo Genius.

Cor	Frequência (Hz)
`redTone`	310
`blueTone`	209
`yellowTone`	252
`greenTone`	415
`errorTone`	42

Problema 1

Implemente o jogo Genius. Após a energização ou *reset* da placa o jogo começa com um primeiro par cor/tom obtido aleatoriamente e aguarda que o jogador aperte um botão. Se o botão correto for pressionado, o jogo repete a sequência anterior e acrescenta uma nova cor/tom aleatoriamente. O jogador deve repetir a nova sequência.

- Em caso de erro, o som de erro deve ser tocado e a última cor pressionada (erroneamente) deve acender pelo mesmo tempo.
- O tamanho máximo da sequência fica a critério do programador. No jogo original apenas 31 cores/tons são possíveis na sequência. Se o jogador consegue acertar a sequência, cores/tons de vitória são emitidos. Faça seu próprio sinal vitória (ver observação mais abaixo).
- O botão *reset* funciona exatamente como o da placa e não requer programação.
- Caso o jogador demore muito para escolher uma cor ele perde o jogo.
- Para dar início a um novo jogo após a derrota ou a vitória, é necessário pressionar o botão de *reset*.
- Use *arrays*!

Cada vez que o jogo é iniciado, espera-se que uma sequência aleatória diferentes seja gerada.
Obs.: Mais detalhes sobre os sons e tempos podem ser vistos em Reverse Engineering an MB Electronic Simon Game.