Programação em Arduino

Prof. Daniel R. Cassar and James Moraes de Almeida

Capacitação em Arduino

Contents

1 Módulo 1 **1**
1.1 Qual é a linguagem de programação do Arduino? 1
1.2 Estrutura básica de um código de Arduino 2
1.3 Portas digitais do Arduino e a função `pinMode` 3
1.4 Portas digitais de `OUTPUT` e a função `digitalWrite` 4
1.5 Fazendo o Arduino esperar com a função `delay` 5
1.6 A constante `LED_BUILTIN` 6
1.7 Criando variáveis inteiras 7
1.8 Operadores aritméticos 9
1.9 Operadores relacionais e variáveis booleanas 10
1.10 Estrutura de controle condicional com `if` 10
1.11 Operadores lógicos 12

1 Módulo 1

1.1 Qual é a linguagem de programação do Arduino?

Para efetivamente usar o Arduino, é necessário programar o seu microprocessador de forma que ele execute as tarefas do seu interesse. O Arduino IDE é o programa que faz a ponte entre o código de programação que nós escrevemos e a compilação e envio deste código para o microprocessador do Arduino.

A linguagem de programação que o Arduino IDE usa é o `C++` com um certo "tempero" extra. A linguagem `C++` é praticamente um superconjunto da linguagem `C`[1]. O "tempero" extra é um conjunto de funções e variáveis características para as diversas comunicações disponíveis no Arduino (portas

[1] Isto é, um código em `C` quase sempre pode ser "entendido" por um compilador de `C++`.

digitais, portas analógicas, comunicação serial, etc. . .).

1.2 Estrutura básica de um código de Arduino

Todo código de Arduino deve conter pelo menos dois blocos: o bloco de configuração e o bloco de laço de repetição. Veja abaixo um exemplo do código para escrever estes dois blocos.

```
1  void setup() {
2      // aqui fica o bloco de configuração
3  }
4
5  void loop() {
6      /* aqui fica o
7         bloco de laço de repetição */
8  }
```

O bloco de configuração (`void setup`) é executado sempre que o Arduino é ligado. Este bloco é executado apenas uma vez durante o funcionamento do Arduino. Ele apenas será executado novamente caso o Arduino desligue e ligue ou caso o botão de *reset* seja pressionado (usualmente é um botão vermelho na placa do Arduino).

O bloco de laço de repetição (`void loop`) é executado após a execução do bloco de configuração. Ao fim da execução do bloco de laço de repetição, o Arduino volta a executar o bloco de laço de repetição. Ou seja, o Arduino fica em um *loop* infinito executando o bloco de laço de repetição até ele ser desligado ou resetado.

Perceba que o código para definir o bloco de configuração é `void setup` seguido de parênteses "()" e chaves "{}". Veremos mais adiante que esta sintaxe está na verdade definindo uma função chamada `setup`. Todo código que está dentro destas chaves é o código que pertence à função `setup` (chamamos isso de bloco). A palavra `void` (que vem do inglês e significa "vazio") está indicando que a função `setup` não tem nenhum valor de retorno. O fato que não existe nenhuma informação dentro dos parênteses nos indica que a função `setup` não requer nenhum argumento[2]. Não se preocupe, isso fará

[2]Na verdade, neste caso podemos omitir os parênteses, mas para manter uma mesma estrutura ao longo do texto os parênteses serão sempre escritos.

mais sentido após ler o tópico sobre funções mais adiante. Por hora apenas certifique-se que a sintaxe do seu código está correta seguindo o exemplo apresentado.

O bloco de laço de repetição segue a mesma regra de sintaxe do bloco de configuração e é escrito como `void loop`.

Note que no código acima temos também dois comentários: "aqui fica o bloco de configuração" e "aqui fica o bloco de laço de repetição". Comentários não são comandos enviados ao Arduino, mas sim informações deliberadamente inseridas no código para documentá-lo. Podemos escrever comentários de duas maneiras diferentes:

1. Usando "`//`", também conhecido como comentário em linha. Neste caso o comentário se encerra quando a linha chega ao seu final.

2. Usando "`/*`" e "`*/`", também conhecido como comentário em bloco. Neste caso todo o texto entre "`/*`" e "`*/`" faz parte do comentário, podendo conter quantas linhas forem necessárias.

Em geral, é melhor ter um código com comentários do que um código sem comentários. Fique à vontade para usar o comentário em linha ou o comentário em bloco de acordo com suas necessidades e preferências.

1.3 Portas digitais do Arduino e a função `pinMode`

As portas digitais do Arduino podem atuar de duas maneiras: enviando sinal (`OUTPUT`) ou recebendo sinal (`INPUT`). Antes de usar qualquer porta digital, é necessário informar o Arduino que esta porta será utilizada. A função para fazer isso é o `pinMode` que em tradução livre significa "modo da pinagem" em português. A função `pinMode` requer dois argumentos: o número da porta digital que será utilizada e o modo de operação (`INPUT` ou `OUTPUT`).

Vamos supor que queremos montar um circuito onde a porta 13 do Arduino irá enviar um sinal eletrônico e a porta 7 irá receber um sinal eletrônico. Podemos escrever isso da seguinte maneira:

```
1  void setup() {
2      pinMode(13, OUTPUT);
3      pinMode(7, INPUT);
4  }
5
6  void loop() {
7  }
```

Observe que o código que configura as portas do Arduino está dentro do bloco de configuração (faz sentido, não é?). Observe também que para usar a função `pinMode` precisamos de parenteses; não apenas isso, mas é dentro dos parênteses que indicamos os argumentos da função. No caso específico da função `pinMode`, o primeiro argumento é o número da porta digital e o segundo argumento é seu modo de operação (`INPUT` ou `OUTPUT`).

Existem mais dois detalhe no código acima, você consegue identificá-los?

O primeiro detalhe é o "`;`" no final das linhas 2 e 3. Cada comando em `C++` deve ser encerrado com um ponto e vírgula. Esta é uma regra de sintaxe desta linguagem de programação. Se você esquecer de colocar todos os pontos e vírgulas necessários seu código não irá compilar.

O segundo detalhe é que as linhas 2 e 3 estão *indentadas*, isto é, elas têm um certo número de espaços antes do texto (4 espaços neste caso). Indentação não é uma regra de sintaxe da linguagem `C++`, logo seu código irá compilar sem problemas caso você não o indente. No entanto, códigos devidamente indentados são mais fáceis de ler e por isso é altamente recomendado que você indente seu código. Você pode indentar o código usando quantos espaços quiser, é fortemente recomendado que sua indentação seja consistente dentro do mesmo arquivo.

1.4 Portas digitais de `OUTPUT` e a função `digitalWrite`

Portas digitais que foram configuradas como `OUTPUT` operam em dois modos diferentes: enviando sinal elétrico (`HIGH`) e não enviando sinal elétrico (`LOW`). Se estas portas estiverem trabalhando no modo `HIGH`, o sinal elétrico enviado é uma voltagem que pode ser 5V ou 3.3V, dependendo do modelo do Arduino. Se, ao contrário, estiverem operando no modo `LOW` elas não enviam sinal elétrico algum (0V).

Uma porta digital `OUTPUT` sempre é iniciada no modo `LOW` quando a função `pinMode` é usada. Para alterar o modo de operação usamos a função `digitalWrite`. Esta função tem dois argumentos: o primeiro é o número da porta e o segundo é o modo de operação (`HIGH` ou `LOW`).

O código abaixo define a porta digital 5 como sendo `OUTPUT` e altera seu modo de operação para `HIGH`, fazendo com que a porta 5 envie sinal elétrico (este sinal pode alimentar um componente eletrônico, por exemplo).

```
1  void setup() {
2      pinMode(5, OUTPUT);
3      digitalWrite(5, HIGH);
4  }
5
6  void loop() {
7  }
```

1.5 Fazendo o Arduino esperar com a função `delay`

O Arduino não tem uma unidade de processamento considerada poderosa nos dias de hoje. Isso quer dizer que ele executa menos tarefas por segundo do que um processador moderno de computador. Mesmo assim, a quantidade de operações que um Arduino pode fazer por segundo é um número consideravelmente grande. Por conta disso, em diversos momentos é necessário comandar explicitamente o Arduino para que ele aguarde um certo tempo antes de prosseguir com a execução do código.

Veja o exemplo abaixo. Ele está declarando que a porta digital 6 é uma porta tipo `OUTPUT` (irá enviar um sinal elétrico) no bloco de configuração. No bloco de laço de repetição o programa está alterando o modo de operação da porta 6 para `HIGH` e depois para `LOW`, efetivamente ligando e desligando o sinal enviado pela porta 6. Suponha que na porta 6 esteja ligado um LED, como você acha que ele irá se comportar?

```
1  void setup() {
2      pinMode(6, OUTPUT);
3  }
```

```
4
5  void loop() {
6      digitalWrite(6, HIGH);
7      digitalWrite(6, LOW);
8  }
```

Se a intenção do código acima era de fazer o LED acender e apagar então temos um problema. O problema é que o Arduino executa as linhas 6 e 7 muito rápido em relação a nossa percepção do tempo. Para fazer o LED efetivamente piscar precisamos de uma forma de pedir para o Arduino aguardar um momento entre os comandos. A função que faz isso é a `delay` e ela tem apenas um argumento: a quantidade de microsegundos que o Arduino deve aguardar (só para relembrar, 1000 microsegundos equivale a 1 segundo). Veja o uso desta função no código abaixo.

```
 1  void setup() {
 2      pinMode(6, OUTPUT);
 3  }
 4
 5  void loop() {
 6      digitalWrite(6, HIGH);
 7      delay(1000);
 8      digitalWrite(6, LOW);
 9      delay(1000);
10  }
```

1.6 A constante `LED_BUILTIN`

Diversos modelos de Arduino têm um LED embutido na própria placa. Este LED é controlado por uma das portas digitais do Arduino. Em diversos modelos, é a porta digital 13 que controla este LED, mas isso não é uma regra! Para facilitar, ao invés de usar o número da porta podemos usar a constante chamada `LED_BUILTIN` que armazena o número na porta digital do LED embutido do Arduino. Veja abaixo um exemplo de código que liga o LED embutido do Arduino.

```
1  void setup() {
2      pinMode(LED_BUILTIN, OUTPUT);
3      digitalWrite(LED_BUILTIN, HIGH);
```

```
4  }
5
6  void loop() {
7  }
```

1.7 Criando variáveis inteiras

Em diversas situações em programação é interessante armazenar uma informação que possa ser utilizada em um momento futuro. Observe o código abaixo.

```
 1  void setup() {
 2      pinMode(6, OUTPUT);
 3  }
 4
 5  void loop() {
 6      digitalWrite(6, HIGH);
 7      delay(1000);
 8      digitalWrite(6, LOW);
 9      delay(1000);
10  }
```

Suponha que ao invés de usar a porta digital 6 você queira usar a porta digital 10 para realizar a tarefa do código acima, como você teria que alterar o código? A resposta é simples: você teria que alterar em todas as linhas que estão relacionadas com a porta digital 6 (linhas 2, 6 e 8) e trocar o número 6 pelo número 10. Seu código ficaria assim:

```
 1  void setup() {
 2      pinMode(10, OUTPUT);
 3  }
 4
 5  void loop() {
 6      digitalWrite(10, HIGH);
 7      delay(1000);
 8      digitalWrite(10, LOW);
 9      delay(1000);
10  }
```

Neste caso tivemos "sorte" pois precisamos alterar apenas 3 linhas. Imagina se tivéssemos que alterar 10 linhas... ou 100 linhas... além de ser trabalhoso é um processo propenso a erros!

Uma forma de evitar este problema é armazenando a informação do número da porta digital em uma variável. Uma variável nada mais é do que um nome que nós atribuímos a uma informação, neste caso a informação é o número da porta. Veja o código abaixo.

```
 1  int pino_saida = 10;
 2
 3  void setup() {
 4      pinMode(pino_saida, OUTPUT);
 5  }
 6
 7  void loop() {
 8      digitalWrite(pino_saida, HIGH);
 9      delay(1000);
10      digitalWrite(pino_saida, LOW);
11      delay(1000);
12  }
```

Na primeira linha do código temos a declaração da variável chamada `pino_saida`. Esta variável foi declarada como sendo do tipo `int` e a ela foi atribuído o valor de 10. Toda variável declarada em `C++` requer um tipo. Existem diversos tipos de variáveis, porém neste momento vamos nos focar no tipo `int` que significa que a variável é um número inteiro entre –32.768 e 32.767.

Observe que a declaração da variável `pino_saida` acontece fora do bloco de configuração e fora do bloco de laço de repetição. Neste caso, dizemos que a variável `pino_saida` foi declarada no escopo global. Variáveis declaradas no escopo global podem ser usadas tanto no bloco de configuração quanto no bloco de laço de repetição. Variáveis declaradas dentro do bloco de configuração só podem ser usadas dentro do bloco de configuração. De forma similar, variáveis declaradas dentro do bloco de laço de repetição só podem ser usadas dentro do bloco de laço de repetição. Por conta disso, é importante se atentar em qual escopo sua variável foi declarada para evitar problemas.

1.8 Operadores aritméticos

Temos a nossa disposição operadores aritméticos como soma (`+`), subtração (`-`), multiplicação (`*`) e divisão (`/`). Veja abaixo um exemplo do uso destes operadores

```
 1  int variavel;
 2
 3  void setup() {
 4      // soma, variavel agora armazena o valor 11
 5      variavel = 10 + 1;
 6      // subtração, variavel agora armazena o valor 5
 7      variavel = 10 - 5;
 8      // multiplicação, variavel agora armazena o valor 25
 9      variavel = variavel * variavel;
10      // divisão, variavel agora armazena o valor 1
11      variavel = variavel / 25;
12  }
13
14  void loop() {
15      // a cada loop aumentamos o valor da variavel em 1
16      variavel = variavel + 1;
17  }
```

No código acima nós criamos uma nova variável com o nome `variavel`. Observe que não é necessário atribuir um valor ao criar uma variável, é apenas necessário indicar o tipo (que no caso é um inteiro). Dentro do bloco de configuração nós atribuímos e alteramos o valor desta variável. Nas linhas 4 e 5 o novo valor da variável foi atribuído usando uma expressão com o operador de soma (`10 + 1`) e de subtração (`10 - 5`). Não tem problema algum alterar os valores de variáveis, faça isso sempre que julgar necessário.

As linhas 9, 11 e 16 são particularmente interessantes: nelas o novo valor da variável depende do valor atual da variável! Esta é uma sintaxe perfeitamente válida em programação[3].

[3]Ela é válida pois o sinal de igual não significa a igualdade matemática, mas sim é um operador de atribuição. O sinal de igual atribui um valor à uma variável.

1.9 Operadores relacionais e variáveis booleanas

Além dos operadores aritméticos temos também a nossa disposição operadores relacionas como o operador de "maior que" (`>`), de "menor que" (`<`), de "maior ou igual a" (`>=`), de "menor ou igual a" (`<=`) e finalmente o operador de "comparação de igualdade" (`==`).

Operadores relacionais comparam dois objetos e retornam um valor booleano. Variáveis booleanas podem ter apenas dois valores: verdadeiro (`true`) ou falso (`false`). Exemplos:

- `4 > 3`: retorna `true`
- `5 < 2`: retorna `false`
- `10 == 10`: retorna `true`

1.10 Estrutura de controle condicional com `if`

Uma das estruturas de controle mais utilizada em programação é o condicional. O condicional nos permite executar um bloco de código apenas quando certa condição é atendida. Condicionais são chamados com a instrução `if`[4]. Vamos ver um exemplo

```
 1  int contador = 0;
 2
 3  void setup() {
 4  }
 5
 6  void loop() {
 7      if (contador == 10) {
 8          /* tudo que estiver neste bloco aqui só
 9             irá ser executado quando o contador for
10             igual a 10 */
11      }
```

[4]Significa "se" em inglês.

```
12      contador++;
13  }
```

O condicional `if` requer apenas um argumento: a condição para que o bloco do `if` seja executado. No exemplo acima, o bloco do `if` só será executado quando a variável contador for igual a 10 (o que faz a sua condição ser `true`).

Em certos casos também queremos executar um bloco de código caso a condição do `if` seja `false`. Para isso precisamos incluir o bloco do `else`[5]. Veja o exemplo abaixo.

```
 1  int contador = 0;
 2
 3  void setup() {
 4  }
 5
 6  void loop() {
 7      if (contador == 10) {
 8          /* tudo que estiver neste bloco aqui só
 9             irá ser executado quando o contador for
10             igual a 10 */
11      }
12      else {
13          /* tudo que estiver neste bloco aqui só
14             irá ser executado quando o contador NÃO
15             for igual a 10 */
16      }
17      contador++;
18  }
```

O bloco do `else` não requer uma condição, basta abrir um bloco de chaves e escrever seu código. No entanto, em muitos casos nós queremos que o bloco do `else` cheque uma condição. Veja um exemplo abaixo.

```
 1  int contador = 0;
 2
```

[5]Significa "se não" em inglês.

```
 3  void setup() {
 4  }
 5
 6  void loop() {
 7      if (contador == 10) {
 8          /* tudo que estiver neste bloco aqui só
 9              irá ser executado quando o contador for
10              igual a 10 */
11      }
12      else if (contador > 10) {
13          /* tudo que estiver neste bloco aqui só
14              irá ser executado quando o contador
15              for maior que 10 */
16      }
17      else {
18          /* tudo que estiver neste bloco aqui só
19              irá ser executado quando o contador NÃO
20              for igual a 10 e NÃO for maior que 10 */
21      }
22      contador++;
23  }
```

Podemos adicionar quantos blocos `else if` quisermos, desde que eles venham depois do bloco `if` e antes do bloco `else` (sendo este último opcional).

1.11 Operadores lógicos

Operadores lógicos de conjunção (`&&`) e disjunção (`||`) nos ajudam a criar condições mais complexas para o `while`, `for` e `if`.

O operador de conjunção retorna o valor `true` apenas quando os valores lógicos das condições na qual opera são ambos `true`, do contrário este operador retorna o valor `false`. Veja o exemplo abaixo.

```
1  int contador = 0;
2
3  void setup() {
```

```
 4  }
 5
 6  void loop() {
 7      if (contador >= 10 && contador <= 20) {
 8          /* tudo que estiver neste bloco aqui só
 9             irá ser executado quando o contador for
10             maior ou igual a 10 e menor ou igual a 20 */
11      }
12      contador++;
13  }
```

O operador de disjunção retorna o valor `true` quando pelo menos um dos valores lógicos das condições na qual opera também é `true`, do contrário este operador retorna o valor `false`. Veja o exemplo abaixo.

```
 1  int contador = 0;
 2
 3  void setup() {
 4  }
 5
 6  void loop() {
 7      if (contador <= 10 || contador >= 20) {
 8          /* tudo que estiver neste bloco aqui só
 9             irá ser executado quando o contador for
10             menor ou igual a 10 ou maior ou igual a 20 */
11      }
12      contador++;
13  }
```