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 1
- 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

2 Módulo 2 **6**
- 2.1 Criando variáveis inteiras 6
- 2.2 Operadores aritméticos 8
- 2.3 Operadores relacionais e variáveis booleanas 9
- 2.4 Laço de repetição com `while` 10
- 2.5 Laço de repetição com `for` 11
- 2.6 Estrutura de controle condicional com `if` 12
- 2.7 Operadores lógicos 14
- 2.8 Portas digitais de `INPUT` e a função `digitalRead` 15
- 2.9 Portas digitais de `INPUT_PULLUP` 16

3 Módulo 3 **17**
- 3.1 Definindo funções no Arduino 17
- 3.2 Criando variáveis reais 20
- 3.3 Portas analógicas do Arduino e a função `analogRead` 21
- 3.4 Comunicação serial entre o Arduino e o computador 21
- 3.5 Criando variáveis inteiras tipo `long` 23
- 3.6 Gerando números aleatórios com a função `random` 23
- 3.7 Portas digitais PWM e a função `analogWrite` 24

4 Módulo 4 **26**
4.1 Usando bibliotecas 26
4.2 Sensor de umidade do solo (LM393) 27
4.3 Sensor de distância ultrassônico (HC-SC04) 28
4.4 Sensor de temperatura e umidade (DHT11) 30
4.5 Comunicação tipo I^2C 32
4.6 Sensor de temperatura e pressão (BMP280) 34
4.7 Lendo e armazenando dados enviados pela comunicação serial usando Python 36

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 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
```

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

```
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 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

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

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 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  }
```

2 Módulo 2

2.1 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.

2.2 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      variavel = 10 + 1;               // soma, variavel agora armazena o valor 11
 5      variavel = 10 - 5;               // subtração, variavel agora armazena o valor
 6      variavel = variavel * variavel;  // multiplicação, variavel agora armazena o va
 7      variavel = variavel / 25;        // divisão, variavel agora armazena o valor 1
 8  }
 9
10  void loop() {
11      variavel = variavel + 1;         // a cada loop aumentamos o valor da variavel
12  }
```

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 6, 7 e 11 são particularmente interessantes: nelas o novo valor da variável depende do valor atual da variável! Esta é uma sintaxe perfeitamente válida em programação[3].

2.3 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`

[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.

- `5 < 2`: retorna `false`

- `10 == 10`: retorna `true`

Em geral usamos operadores relacionais para comparar o valor de uma variável com um valor de uma constante ou para comparar o valor de uma variável com o valor de outra variável.

2.4 Laço de repetição com `while`

O bloco de laço de repetição do Arduino (`void loop`) nos fornece uma maneira simples de indicar para o Arduino quais linhas de código ele deve repetir. Em certos casos, apenas o bloco de laço de repetição não é suficiente e precisamos de mais controle. O comando `while`[4] é uma forma de criar laços de repetição com condições controladas. Observe o exemplo abaixo.

```
 1  int contador = 0;
 2
 3  void setup() {
 4  }
 5
 6  void loop() {
 7      while (contador < 100) {
 8          /* tudo que estiver neste bloco aqui
 9             irá ser executado 100 vezes */
10          contador = contador + 1;
11      }
12
13      contador = 0;  // Aqui estamos reinicializando o contador em zero
14
15      /* tudo que estiver fora do ‘while‘ não é afetado por ele.
16         Enquanto o bloco dentro do ‘while‘ rodou 100 vezes,
17         este código fora do ‘while‘ irá rodar apenas 1 vez. */
18  }
```

[4]Significa “enquanto” em inglês.

Neste código temos o comando `while` seguido de um parênteses e um bloco de código entre chaves. O código dentro do parênteses é a chamada *condição* do `while`. O bloco de código entre chaves do comando `while` apenas será executado se a condição for verdadeira (`true`). Neste caso estamos checando se a variável `contador` é menor que a constante 100. Cada vez que o bloco de código do `while` se encerra, o Arduino retorna ao início do bloco do `while` e checa a condição novamente. Se a condição permanecer verdadeira o bloco do `while` é executado novamente, do contrário o bloco do `while` não é executado e o programa segue para a linha imediatamente após o final do bloco do `while`.

A variável `contador` do código acima é o que chamamos de *variável acumuladora*. Veja que é responsabilidade desta variável "contar" quantas vezes o bloco do `while` foi executado. Toda vez que o bloco é executado, esta variável é acrescida do número 1 (linha 10). O processo de somar 1 a uma variável é uma tarefa tão usada em programação que é comum existir um "açúcar sintático" para expressar essa tarefa. Em `C++` podemos reescrever a linha 10 do código acima da seguinte maneira: "`contador++`" (ou seja, o nome da variável com um "`++`" no final).

Se a condição do `while` for sempre verdadeira, então o bloco do `while` entrou em um *loop* infinito. Se a condição do `while` nunca for verdadeira, então o bloco do `while` jamais será executado. Para evitar problemas, é usual zerar o contador logo após sair do bloco do `while`. Isto está sendo feito na linha 13 do código acima.

2.5 Laço de repetição com `for`

Outro laço de repetição disponível é o `for`[5]. O laço `for` tem uma sintaxe um pouco diferente da sintaxe do laço `while`, sendo que ele próprio "cuida" da sua variável acumuladora. Vamos ver o código anterior reescrito usando um laço `for`:

```
1  void setup() {
2  }
3
4  void loop() {
```

[5]Significa "para" em inglês.

```
 5      for (int contador = 0, contador < 100, contador++) {
 6          /* tudo que estiver neste bloco aqui
 7             irá ser executado 100 vezes */
 8      }
 9
10      /* tudo que estiver fora do ‘for‘ não é afetado por ele.
11         Enquanto o bloco dentro do ‘for‘ rodou 100 vezes,
12         este código fora do ‘for‘ irá rodar apenas 1 vez. */
13  }
```

Observe que a instrução `for` requer três argumentos para ser executado.

1. O primeiro argumento é a *inicialização*. Ele é o primeiro comando do laço `for` a ser executado e só é executado uma vez ao entrar no laço. É aqui que nós criamos a nossa variável acumuladora para ser usada no laço `for`.

2. O segundo argumento é a *condição*. Ele é similar à condição que vimos no laço `while`. O bloco do laço `for` será executado se a condição for `true` e não será executado se a condição for `false`.

3. O terceiro argumento é o *incremento*. Ele é executado toda vez que a condição for `true` e serve para atualizar a variável acumuladora.

2.6 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`[6]. Vamos ver um exemplo

```
1  int contador = 0;
2
3  void setup() {
4  }
```

[6]Significa “se” em inglês.

```
 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      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`[7]. 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  }
```

[7] Significa "se não" em inglês.

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
 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).

2.7 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 oper-

ador 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  }
```

2.8 Portas digitais de `INPUT` e a função `digitalRead`

Portas digitas que foram configuradas como `INPUT` servem para receber sinal elétrico. Para checar se estas portas estão recebendo sinal elétrico usamos a função `digitalRead`. Esta função requer apenas um argumento: o número

da porta digital.

Portas digitais que foram configuradas como `INPUT` são usadas para checar se o Arduino está recebendo um sinal elétrico. A função usada para checar se a porta está recebendo sinal é a `digitalRead`[8]. Esta função recebe como argumento o número da porta digital e pode retornar dois valores diferentes: `HIGH` caso a porta esteja recebendo sinal elétrico e `LOW` caso a porta não esteja recebendo sinal elétrico.

O código abaixo é um exemplo de como usar a função `digitalRead` para checar se a porta digital 8 está recebendo sinal elétrico ou não (e executar tarefas diferentes para cada caso).

```
 1  void setup() {
 2      pinMode(8, INPUT);
 3  }
 4
 5  void loop() {
 6      if (digitalRead(8) == HIGH) {
 7          /* Este bloco só sera executado se a porta digital
 8             8 estiver recebendo sinal elétrico */
 9      }
10      else {
11          /* Este bloco só sera executado se a porta digital
12             8 não estiver recebendo sinal elétrico */
13      }
14  }
```

2.9 Portas digitais de `INPUT_PULLUP`

Além de `INPUT` e `OUTPUT`, existe um terceiro modo de operação das portas digitais do Arduino: o modo `INPUT_PULLUP`. Este modo opera de maneira similar ao modo `INPUT`, ou seja, ele serve para checar se a porta está recebendo sinal elétrico ou não (e para realizar essa leitura usamos a mesma função `digitalRead` apresentada na seção anterior). A diferença é que no modo `INPUT_PULLUP` a porta digital atua como um resistor de pull-up: a porta constantemente retorna um sinal `HIGH` quando nada está conectado e

[8] Significa "leitura digital" em tradução livre do inglês.

retorna um sinal `LOW` quando a porta está aterrada.

3 Módulo 3

3.1 Definindo funções no Arduino

É comum em programação o conceito de código reutilizável: trata-se de um conjunto de comandos que, após definidos, podem ser utilizados em outros locais. Nós já vimos algumas funções embutidas do Arduino: `pinMode`, `digitalWrite`, `digitalRead` e `delay` são alguns exemplos. Estas funções já foram escritas para nós e podemos usar a hora que quisermos. Imagine se cada vez que você fosse configurar uma porta digital você tivesse que escrever todos os comandos da função `digitalWrite`... seria bastante contraprodutivo!

Nós podemos definir nossas próprias funções. Na realidade, todo código de Arduino requer a definição de duas funções em particular, uma chamada `setup` e outra chamada `loop`. A sintaxe de definição destas funções nós já vimos no Módulo 1 e está repetida abaixo.

```
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  }
```

As funções `setup` e `loop` não recebem nenhum argumento[9] e também não retornam nenhuma informação. Nós observamos que estas funções não retornam nenhuma informação pela presença da palavra `void` antes do nome da função. Veja abaixo um exemplo da definição e uso de uma função que retorna um valor inteiro.

```
1  int armazena_numero;
```

[9] Alguns livros chamam argumentos de parâmetros. Aqui usaremos o termo argumento.

```
 2
 3  void setup() {
 4      armazena_numero = retorna_oito();
 5  }
 6
 7  void loop() {
 8  }
 9
10  int retorna_oito() {
11      return 8;
12  }
```

Após rodar este programa, qual você acha que é o valor armazenado na variável `armazena_numero`? Vamos por partes:

1. A variável `armazena_numero` é definida na linha 1 como sendo uma variável inteira, mas sem nenhum valor atribuído a ela.

2. Na linha 4, dentro do bloco de configuração, a variável `armazena_numero` recebe o valor da função `retorna_oito`. Sabemos que `retorna_oito` é uma função pois funções são sempre chamadas com os argumentos dentro de parênteses. Neste caso, como a função tem zero argumentos, não temos nada dentro dos parênteses (mas mesmo assim precisamos escrever os parênteses!). Mas onde está a definição da função `retorna_oito`?

3. A função `retorna_oito` é definida na linha 10. Não tem nenhum problema definir funções após o `void setup` e o `void loop`, mesmo que estas funções sejam chamadas dentro destes blocos de código. Nós sabemos que essa é a definição de uma função e não de uma variável por conta da presença de chaves indicando um bloco de código (isso não existe na definição de variáveis!). Também sabemos que esta função retorna um valor inteiro ao ser executada por conta do `int` antes do nome da função. Se ela não retornasse nada teríamos um `void` no lugar do `int`.

4. Dentro da função `retorna_oito` nós temos um bloco de código delimitado por chaves. Este bloco tem apenas uma linha de código, mas

poderia ter quantas fossem necessárias. Vimos acima que esta função tem um retorno; o comando para retornar um valor é o `return` (que significa retornar em inglês). Neste caso específico, a função retorna o valor 8.

5. Sabendo o que acontece quando a função `retorna_oito` é executada, agora podemos responder a pergunta: o valor armazenado na variável `armazena_numero` ao final da execução do programa será 8.

A função `retorna_oito` definida acima não é muito interessante... ela apenas retorna o valor 8. Mas o importante neste momento é entender como funções com retorno são definidas. Para tornar funções mais interessantes, vamos ver no exemplo abaixo como definir funções que recebem argumentos.

```
1  int armazena_numero;
2
3  void setup() {
4      armazena_numero = multiplica_por_dois(10);
5      acende_led(LED_BUILTIN, 1000);
6  }
7
8  void loop() {
9  }
10
11 int multiplica_por_dois(int numero) {
12     return numero * 2;
13 }
14
15 void acende_led(int porta, int tempo_aceso_ms) {
16     pinMode(porta, HIGH);
17     delay(tempo_aceso_ms);
18     pinMode(porta, LOW);
19 }
```

Duas funções foram definidas no código acima: `multiplica_por_dois` na linha 11 e `acende_led` na linha 15. Observe que na definição delas temos um parênteses antes de abrir as chaves do bloco de código da função. É dentro deste parênteses que definimos os argumentos das funções, sempre

indicando o tipo do argumento e o nome dele. No caso da função ter dois ou mais argumentos, estes são separados por vírgulas.

A função `multiplica_por_dois` recebe um argumento inteiro e retorna o valor da multiplicação deste argumento pelo número 2. Esta função é usada na linha 4 para atualizar o valor da variável `armazena_numero`, que a partir de agora armazena o valor 20.

A função `acende_led` recebe dois argumentos, ambos inteiros, e não retorna nenhum valor. Esta função envia sinal elétrico por uma porta do Arduino (argumento `porta`) por um certo tempo em milisegundos (argumento `tempo_aceso_ms`) e após isso para de enviar o sinal elétrico por esta porta. Esta função é usada na linha 5 do bloco de configuração para acender o LED embutido do Arduino por 1000 milisegundos.

3.2 Criando variáveis reais

Variáveis inteiras têm duas limitações importantes: não permitem números decimais e estão restritas a números na faixa de –32.768 a 32.767. Existem casos onde precisamos armazenar um número real em uma variável que não tenha essas limitações.

As variáveis tipo ponto flutuante (`float`) podem armazenar números decimais e valores desde $-3{,}40 \times 10^{38}$ até $3{,}40 \times 10^{38}$.

A sintaxe para criar variáveis do tipo `float` é similar à sintaxe para criar variáveis do tipo `int`. Veja no exemplo abaixo.

```
1  float variavel_float = 3.1415;
2  float variavel_com_notacao_e = 1.23456e4; // equivale a 12345.6
3
4  void setup() {
5  }
6
7  void loop() {
8  }
```

3.3 Portas analógicas do Arduino e a função `analogRead`

As portas digitas podem receber ou enviar apenas dois sinais distintos: `LOW` ou `HIGH`[10]. Já as portas analógicas do Arduino podem apenas *receber* sinal, porém não estão limitadas em apenas dois sinais distintos: elas recebem um sinal que é um valor entre 0 e 1023. Ter um maior intervalo de valores é particularmente interessante quando temos interesse em sensores que medem quantidades contínuas como temperatura, pressão e umidade.

As portas analógicas do Arduino são enumeradas e começam com a letra A[11]. Para ler o sinal elétrico que chega em uma porta analógica usamos a função `analogRead`. Esta função tem apenas um argumento, o número da porta analógica. Veja um exemplo abaixo.

```
 1  int porta_analogica = A3;  // observe a letra "A" na frente do número
 2  int valor_lido;
 3
 4  void setup() {
 5  }
 6
 7  void loop() {
 8      valor_lido = analogRead(porta_analogica);
 9      delay(1000);
10  }
```

Neste exemplo, dentro do bloco de laço de repetição a variável `valor_lido` recebe o valor lido na porta analógica A3 (que será um número inteiro de 0 a 1023). Após ler e armazerar o valor, o código aguarda 1 segundo e o laço se repete.

3.4 Comunicação serial entre o Arduino e o computador

Um Arduino pode enviar e receber informação de um computador através de sua comunicação serial. Para isso, o primeiro passo é o Arduino estar

[10] Falso ou verdadeiro, desligado ou ligado, zero ou um. Portas digitais são portas binárias.

[11] Em algumas placas, certas portas digitais funcionam também como portas analógicos. Para mais informações acesse o link `https://www.arduino.cc/reference/en/language/functions/analog-io/analogread/`

conectado ao computador por um cabo USB.

A comunicação serial é controlada pelo objeto `Serial` e precisa ser explicitamente inicializada antes do uso. Veja o exemplo abaixo.

```
 1  int porta_analogica = A3;  // observe a letra "A" na frente do número
 2  int valor_lido;
 3
 4  void setup() {
 5      Serial.begin(9600);
 6  }
 7
 8  void loop() {
 9      valor_lido = analogRead(porta_analogica);
10      Serial.print(valor_lido);
11      delay(1000);
12  }
```

A linha 5 do código acima é a inicialização da comunicação serial entre o Arduino e o computador. Observe que a função `begin`[12] faz parte do objeto `Serial`, sendo que para chamá-la é necessário primeiro escrever o nome do objeto seguido de um ponto (`Serial.begin`). Esta função tem apenas um argumento, sendo ele a velocidade da comunicação (em bits por segundo). Na maioria dos casos a velocidade de 9600 é a recomendada.

A linha 10 do código acima é um dos comandos disponíveis para enviar informação do Arduino para o computador. A função se chama `print`[13] e ela faz parte do objeto `Serial`, logo deve ser chamada usando a sintaxe `Serial.print`. Esta função envia para o computador o seu argumento em formato de texto. No exemplo, o argumento é o valor lido na porta analógica A3.

O mesmo código acima pode ser escrito usando a função `Serial.println` no lugar da função `Serial.print`. A diferença é que a função `Serial.println`

[12] Significa "iniciar" em inglês.

[13] Em tradução livre do inglês seria "imprimir". A palavra print em programação não significa literalmente imprimir um texto usando uma impressora. Usualmente a palavra print indica que alguma informação será mostrada para o usuário de forma visual no seu monitor. No caso do Arduino, os comandos tipo `print` enviam informação para o monitor da porta serial.

automaticamente inicia uma nova linha após executar o print e a função `Serial.print` não faz isso de forma automática.

3.5 Criando variáveis inteiras tipo `long`

Variáveis inteiras declaradas com `int` podem armazenar qualquer número inteiro entre –32.768 e 32.767. Existe um outro tipo de variável inteira que é capaz de armazenar números muito maiores, trata-se das variáveis tipo `long`. Estas variáveis são capazes de armazerar qualquer inteiro entre –2.147.483.648 e 2.147.483.647. A declaração de variáveis tipo `long` é similar à declaração de variáveis tipo `int`, veja um exemplo no código abaixo.

```
1  int variavel_tipo_int;
2  long variavel_tipo_long;
3
4  void setup() {
5  }
6
7  void loop() {
8  }
```

3.6 Gerando números aleatórios com a função `random`

A função `random` do Arduino é usada para sortear números aleatórios. Na verdade, ela é usada para gerar números *pseudoaleatórios*. A realidade é que é bastante difícil sortear números de forma perfeitamente aleatória usando computadores e na grande maioria dos casos nós trabalhamos com um sorteador que finge ser perfeitamente aleatório mas efetivamente não o é (por isso são pseudoaleatórios). Esse fato costuma estar implícito em textos de programação. Então não ficamos sempre dizendo que são números pseudoaleatórios, mas sim avisamos quando for o caso dos números serem números aleatórios de fato.

A função `random` pode receber um ou dois argumentos. Se receber apenas um argumento, então ela irá sortear um número inteiro entre zero e o valor do argumento subtraído de 1. Se receber dois argumentos, então ela irá sortear um número inteiro entre o primeiro argumento e o segundo argumento subtraído de 1. Observação: se o valor da função `random` for armazenado em

uma variável, ela deve ser do tipo `long`.

Associado ao processo de gerar números aleatórios existe o conceito da "semente aleatória". Este conceito está intimamente relacionado ao fato de estarmos trabalhando com números pseudoaleatórios. Esta "semente aleatória" serve para dar o pontapé inicial na função `random` para ela iniciar seus sorteios. Toda a sequência de números sorteados depende desta semente! Se a semente aleatória for sempre o mesmo valor, então a sequência de números produzidos pela função `random` será sempre a mesma. Por conta disso, é interessante sempre escolher uma semente aleatória diferente. A função que configura a semente aleatória é a `randomSeed`. Ela recebe apenas um argumento que é justamente um número inteiro positivo representando a semente aleatória. A forma recomendada na documentação do Arduino para usar a função `randomSeed` é alimentando ela com o valor lido de uma porta analógica que não tenha nenhum componente eletrônico ligado (isto pois existe uma variação inerente no sistema que altera essa leitura de forma imprevisível).

O código abaixo exemplifica tudo que foi apresentado acima.

```
 1  long numero_aleatorio;
 2  int porta_analogica_sem_nada_conectado = A0;
 3
 4  void setup() {
 5      Serial.begin(9600);
 6      randomSeed(analogRead(porta_analogica_sem_nada_conectado));
 7  }
 8
 9  void loop() {
10      numero_aleatorio = random(0, 1001);  // sorteia um número entre 0 e 1000
11      Serial.println(numero_aleatorio);
12      delay(100);
13  }
```

3.7 Portas digitais PWM e a função `analogWrite`

O Arduino não possui saída analógica, suas portas analógicas são apenas para recebimento de sinal. Certas portas digitais do Arduino conseguem *simular* um envio de sinal analógico através da técnica de Modulação por Largura de Pulso (Pulse Width Modulation do inglês, PWM). Esta técnica

consiste em enviar um sinal digital na forma de ondas quadradas de alta frequência. O controle do tempo de permanência na crista e no vale da onda permite simular voltagens entre zero e a voltagem máxima de envio da porta digital do Arduino (3,3 ou 5 Volts, dependendo do modelo).[14]

O envio do sinal PWM é feito com a função `analogWrite`. Esta função tem dois argumentos: o primeiro é o número da porta digital e o segundo é a razão cíclica (duty cycle em inglês). A razão cíclica deve ser um valor entre 0 e 255 e indica a fração da voltagem máxima que será enviada. Por exemplo, um valor de 0 simula o envio de 0% da voltagem máxima (0V), um valor de 255 simula o envio de 100% da voltagem máxima e um valor de 127 simula o envio de 50% da voltagem máxima. Este tipo de sinal PWM pode ser usado para alterar o brilho de um LED e controlar a velocidade de um motor.

Apenas algumas das portas digitais do Arduino permitem o envio de sinal PWM. As portas e as frequências de operação podem ser consultadas na documentação oficial em `https://www.arduino.cc/reference/en/language/functions/analog-io/analogwrite/`. No Arduino Leonardo, por exemplo, as portas com suporte ao envio de sinal PWM são as de número 3, 5, 6, 9, 10, 11 e 13 (todas operando na frequência de 490 Hz, com exceção das portas 3 e 11 que operam na frequência de 980 Hz).

Veja um exemplo do uso da função `analogWrite` abaixo. Neste código, a voltagem enviada na porta 6 sobe gradativamente com o passar do tempo até atingir a voltagem máxima, depois disso repete este processo novamente.

```
 1  int porta = 10;  // Esta é uma porta com suporte a PWM em diversas placas
 2
 3  void setup() {
 4      pinMode(porta, OUTPUT);
 5  }
 6
 7  void loop() {
 8      analogWrite(porta, 0);
 9      delay(1000);
10      analogWrite(porta, 100);
11      delay(1000);
```

[14] Para mais informações veja a documentação em `https://www.arduino.cc/en/pmwiki.php?n=Tutorial/PWM`.

```
12      analogWrite(porta, 150);
13      delay(1000);
14      analogWrite(porta, 200);
15      delay(1000);
16      analogWrite(porta, 255);
17      delay(1000);
18  }
```

4 Módulo 4

4.1 Usando bibliotecas

No contexto de programação, uma biblioteca é um conjunto de código reutilizável que nos ajuda a fazer uma certa tarefa. A vantagem de se usar bibliotecas é que não precisamos "reinventar a roda" cada vez que quisermos fazer uma tarefa que já foi implementada por outra pessoa.

Algumas bibliotecas fazem parte das chamadas "bibliotecas padrão" do Arduino e já vem instaladas "de fábrica" quando se instala o Arduino IDE. A biblioteca para gerenciar a memória EEPROM do Arduino e a biblioteca para comunicação com displays de LCD são dois exemplos de bibliotecas padrão.

É bastante comum a necessidade de usar bibliotecas que realizam tarefas não contempladas nas bibliotecas padrão. Existem duas formas de instalar bibliotecas usando o Arduino IDE:

1. Utilizando o gerenciador de bibliotecas disponível no menu `Sketch / Incluir Biblioteca / Gerenciar Bibliotecas`. Este é o procedimento preferencial caso a biblioteca de interesse esteja disponível no gerenciador. Após abrir o gerenciador, buscar a biblioteca desejada e instalar clicando no botão `Instalar`.

2. Instalando a biblioteca usando arquivo `.zip` disponível no menu `Sketch / Incluir Biblioteca / Adicionar Biblioteca .ZIP`. Este procedimento é necessário quando a biblioteca não está no gerenciador de bibliotecas do Arduino.

Antes de usar as funções e objetos contidos em uma determinada biblioteca é necessário carregar ela usando o comando `#include`. O código abaixo carrega a biblioteca `EEPROM` (isso ocorre na linha 1) e utiliza das funções `EEPROM.length` e `EEPROM.write` que são fornecidas por esta biblioteca. A biblioteca `EEPROM` é usada para ler e escrever a memória EEPROM interna do Arduino. O código abaixo serve para "limpar" a memória EEPROM convertendo todos seus valores para zero.

```
 1  #include <EEPROM.h>
 2
 3  void setup() {
 4      for (int i = 0 ; i < EEPROM.length() ; i++) {
 5          EEPROM.write(i, 0);
 6      }
 7  }
 8
 9  void loop() {
10  }
```

Para saber mais sobre as diversas bibliotecas do Arduino veja a documentação oficial em `https://www.arduino.cc/reference/en/libraries/`.

4.2 Sensor de umidade do solo (LM393)

O sensor de umidade do solo LM393 faz exatamente o que o nome sugere: ele é capaz de quantificar o quão úmido o solo está. Este sensor costuma ser usado para sistemas de rega automática controlados pelo Arduino.

O sensor LM393 tem duas partes: a sonda e o módulo. A sonda tem um formato de "U" e é a parte que entra em contato com o solo. A sonda se conecta com o módulo pelos pinos positivo e negativo do módulo. O módulo têm 4 pinos adicionais, sendo eles:

- A0: saída analógica;
- D0: comunicação digital;

- GND: terra. Deve ser ligado ao GND do Arduino;

- VCC: alimentação. Deve ser alimentado pela saída 5V do Arduino (ou 3.3V, depende do modelo do Arduino).

A comunicação digital retorna apenas o valor 0 ou 1 dependendo da umidade e da configuração do potenciômetro do módulo. Já a saída analógica entrega um valor de 0 até 1023 que está relacionado com a umidade do solo. Em um sensor novo, um valor de leitura menor que 400 indica um solo úmido e um valor maior que 800 indica um solo seco. A resposta do sensor varia com o tempo de uso e costuma durar por volta de um ano de uso contínuo.

O sensor de umidade do solo não requer nenhuma biblioteca para seu uso.

4.3 Sensor de distância ultrassônico (HC-SC04)

O sensor de distância ultrassônico HC-SC04 é capaz de medir distâncias entre 2 centímetros até 4 metros. O sensor tem um emissor e um receptor de ultrassom e a medida é realizada computando o tempo entre o sinal emitido pelo emissor e recebido pelo receptor.

Este sensor tem 4 pinos:

- VCC: alimentação. Ligar na saída 5V do Arduino;

- Trig: ligar em uma porta digital do Arduino. É este pino que controla o envio da onda ultrassônica;

- Echo: ligar em uma porta digital do Arduino. É neste pino que recebemos a informação do recebimento da onda ultrassônica;

- GND: terra. Deve ser ligado ao GND do Arduino;

A operação deste sensor é facilitada com o uso de uma biblioteca. Existem diversas bibliotecas escritas para controlar este sensor. Uma delas pode ser instalada pelo próprio gerenciador de bibliotecas do Arduino IDE: a biblioteca *Ultrasonic* escrita pelo Erick Simões. O código desta biblioteca é aberto e pode ser conferido no Github em `https://github.com/ErickSimoes/Ultrasonic`. Veja um exemplo do uso desta biblioteca abaixo.

```
#include <Ultrasonic.h>

int pino_echo = 13;
int pino_trig = 12;
float leitura;

Ultrasonic sensor_us(pino_trig, pino_echo);  // cria o objeto sensor_us

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

void loop() {
    leitura = sensor_us.read();  // lê a distância em centímetros
    Serial.print("Distancia em cm: ");
    Serial.print(leitura);
    Serial.println();
    delay(1000);
}
```

Assim como já vimos anteriormente, é necessário carregar a biblioteca usando o comando `#include` (linha 1). Depois disso, é necessário criar uma instância da classe `Ultrasonic`. A sintaxe para isso lembra a sintaxe de criação de variáveis e funções:

1. Primeiro é necessário declarar o tipo da classe. Neste caso estamos criando uma classe do tipo `Ultrasonic`. Esta classe é definida na biblioteca que importamos na linha 1.

2. Após declarar o tipo da classe escrevemos o nome do objeto que será criado. Neste caso demos o nome de `sensor_us`.

3. Finalmente, é necessário fornecer os argumentos de inicialização da classe. Para o sensor HC-SC04 precisamos de dois argumentos: a porta digital conectada ao pino Trig e a porta digital conectada ao pino Echo.

O objeto `sensor_us` é usado na linha 14 do código acima. Veja que para usar este objeto nós escrevemos o nome dele, um ponto e por fim o nome da função que desejamos usar. Neste caso é a função `read` que não recebe nenhum argumento e retorna a distância em centímetros.

4.4 Sensor de temperatura e umidade (DHT11)

O sensor de temperatura e umidade DHT11 tem 4 pinos não nomeados. Se você estiver olhando para a face com furinhos e os pinos estiverem apontando para baixo, então da esquerda para a direita temos:

- VCC: alimentação. Ligar na saída 5V do Arduino;
- DADOS: comunicação de dados. Deve ser ligado em uma porta analógica;
- NC: este pino não é utilizado;
- GND: terra. Deve ser ligado ao GND do Arduino;

A biblioteca recomendada para a operação deste sensor é a "DHT sensor library" desenvolvida pela Adafruit. Esta biblioteca pode ser instalada pelo gerenciador de bibliotecas do Arduino IDE. Veja abaixo um exemplo da utilização desta biblioteca.

```
1  #include <DHT.h>
2
3  float leitura_umidade;
4  float leitura_temperatura;
5
```

```
 6  int pino_dados = A1;
 7
 8  DHT sensor_dht(pino_dados, DHT11);
 9
10  void setup() {
11      Serial.begin(9600);
12      sensor_dht.begin();
13  }
14
15  void loop() {
16      leitura_umidade = sensor_dht.readHumidity();
17      leitura_temperatura = sensor_dht.readTemperature();
18
19      Serial.print("Umidade (em porcentagem): ");
20      Serial.print(leitura_umidade);
21      Serial.println();
22
23      Serial.print("Temperatura (Celsius): ");
24      Serial.print(leitura_temperatura);
25      Serial.println();
26
27      delay(1000)
28  }
```

Novamente temos a inclusão da biblioteca com o comando `#include` na linha 1 e a criação de uma instância da classe fornecida por esta biblioteca na linha 8. A classe `DHT` requer dois argumentos para ser instanciada: o primeiro é o pino analógico onde o sensor está conectado e o segundo é o tipo de sensor. Para o sensor DHT11 basta escrever `DHT11` no segundo argumento. Na verdade a variável `DHT11` é definida dentro do código da própria biblioteca e é criada quando a biblioteca é carregada na linha 1.

No exemplo acima, o objeto que foi criado chama-se `sensor_dht`. Ele requer uma inicialização com a função `begin` antes de ser operado (veja a linha 12). Após ser inicializado, a leitura da umidade se dá pela função `readHumidity` e a leitura da temperatura se dá pela função `readTemperature`. Ambas são funções de zero argumentos.

4.5 Comunicação tipo I^2C

O Arduino tem capacidade de controlar sensores através do protocolo de comunicação I^2C. I^2C é um protocolo relativamente simples e barato, porém não é indicado para tarefas que exigem alta velocidade de comunicação. Diversos sensores do Arduino têm suporte para este tipo de comunicação.

I protocolo I^2C utiliza de uma linha de dados (chamada SDA) e um barramento com clock (chamado de SCL). Quando um sensor é controlado pelo Arduino seguindo este protocolo, é necessário conectar os pinos SDA e SCL do sensor nas portas corretas da placa Arduino.

O pino SCL deve estar ligado na porta SCL do Arduino Leonardo, na porta A5 do Arduino Nano ou na porta 21 do Arduino Due. Para outros modelos de Arduino você deve checar qual é a porta SCL da placa.

O pino SDA deve estar ligado na porta SDA do Arduino Leonardo, na porta A4 do Arduino Nano ou na porta 20 do Arduino Due. Para outros modelos de Arduino você deve checar qual é a porta SDA da placa.

Suponha que você acabou de montar um circuito eletrônico contendo um Arduino e pelo menos um sensor que se comunica utilizando o protocolo I^2C. Como checar se a comunicação I^2C está funcionando? Para isso, basta executar o código abaixo[15]. A biblioteca `Wire` importada na linha 1 é uma das bibliotecas embutidas do Arduino, não é necessário nenhuma instalação adicional. Os detalhes da implementação deste código não serão aprofundados aqui.

```
1  #include <Wire.h>
2
3  byte error, address;
4  int nDevices;
5
6  void setup() {
7      Wire.begin();
8      Serial.begin(9600);
9      while (!Serial);
```

[15] Adaptado de i2c_scanner disponível em `https://playground.arduino.cc/Main/I2cScanner/`.

```
10          Serial.println("\nI2C Scanner");
11  }
12
13  void loop() {
14      Serial.println("Escaneando...");
15
16      nDevices = 0;
17      for (address = 1; address < 127; address++) {
18          Wire.beginTransmission(address);
19          error = Wire.endTransmission();
20
21          if (error == 0) {
22              Serial.print("Dispositivo I2C encontrado no endereco 0x");
23              if (address<16)
24                  Serial.print("0");
25              Serial.print(address,HEX);
26              Serial.println("  !");
27              nDevices++;
28          }
29          else if (error==4) {
30              Serial.print("Erro desconhecido no endereco 0x");
31              if (address<16)
32                  Serial.print("0");
33              Serial.println(address,HEX);
34          }
35      }
36      if (nDevices == 0)
37          Serial.println("Nenhum dispositivo I2C encontrado\n");
38      else
39          Serial.println("Fim\n");
40
41      delay(5000);
```

Ao executar este código existem três possibilidades de resultados que podem ser acompanhados pelo monitor serial:

1. A comunicação I^2C entre o Arduino e o sensor está funcionando. Neste caso a informação do endereço de comunicação é exibido;

2. A comunicação I^2C entre o Arduino e o sensor tem algum problema e resultou em um erro de comunicação;

3. Nenhum dispositivo com comunicação I^2C foi encontrado. Se isso acontecer mesmo com o circuito aparantemente montado corretamente, cheque as conexões dos pinos. Pode ser um problema de mau contato!

4.6 Sensor de temperatura e pressão (BMP280)

O sensor de temperatura e pressão BMP280 tem 6 pinos e pode trabalhar em dois tipos de comunicação: I^2C ou SPI. O protocolo SPI é mais rápido que o I^2C, porém no caso específico deste sensor esta vantagem efetivamente não ocorre por conta do tempo necessário do sensor para realizar a medida. Na comunicação tipo I^2C os pinos têm o seguinte significado:

- VCC: alimentação. Ligar na saída 3.3V do Arduino;
- GND: terra. Deve ser ligado ao GND do Arduino;
- SCL: pino serial clock;
- SDA: pino de linha de dados (serial data);
- CSB: pino chip select (não é utilizado se a comunicação for I^2C);
- SDO: pino serial data out;

Os pinos SCL e SDA devem ser conectados nos pinos corretos assim como apresentado na seção anterior.

O pino SDO controla qual o endereço I^2C que será usado durante a comunicação. Se este pino não recebe sinal elétrico então o endereço de comunicação utilizado será o `0x76`. Se este pino está conectado a uma alimentação

de 3.3V, então o endereço de comunicação utilizado será o `0x77`. Você pode testar qual o endereço está sendo utilizado rodando o código apresentado na seção anterior. Diferentes bibliotecas consideram diferentes endereços de comunicação, então fique atento!

A biblioteca recomendada para este sensor é a "Adafruit BMP280 Library" desenvolvida pela Adafruit e disponível no gerenciador de bibliotecas do Arduino IDE. Atenção: esta biblioteca considera que o endereço de comunicação I^2C é o `0x77`. Leve isso em consideração quando for montar seu circuito! Sugestão: antes de rodar o código abaixo, garanta que a comunicação I^2C entre o Arduino e o sensor está funcionando seguindo os passos da seção anterior.

O código abaixo mostra como medir a temperatura e a pressão usando o sensor BMP280. Se atente para a linha 1 onde carregamos a biblioteca do sensor, a linha 3 onde a classe do sensor é definida, a linha 7 onde o sensor é inicializado, as linhas 10-14 onde o sensor é configurado, e as linhas 19 e 22 onde as medidas de temperatura (em Celsius) e pressão (em Pascal) são realizadas.

```
1  #include <Adafruit_BMP280.h>
2
3  Adafruit_BMP280 sensor_bmp;
4
5  void setup() {
6      Serial.begin(9600);
7      sensor_bmp.begin();
8
9      // Configuração do sensor. Apenas altere se souber o que está fazendo
10     sensor_bmp.setSampling(Adafruit_BMP280::MODE_NORMAL,
11                            Adafruit_BMP280::SAMPLING_X2,
12                            Adafruit_BMP280::SAMPLING_X16,
13                            Adafruit_BMP280::FILTER_X16,
14                            Adafruit_BMP280::STANDBY_MS_500);
15 }
16
17 void loop() {
18     Serial.print("Temperatura (Celsius) = ");
19     Serial.println(sensor_bmp.readTemperature());
```

```
20
21      Serial.print("Pressao (Pa) = ");
22      Serial.println(sensor_bmp.readPressure());
23
24      Serial.println();
25      delay(2000);
26  }
```

4.7 Lendo e armazenando dados enviados pela comunicação serial usando Python

O Arduino IDE conta com um monitor da porta serial onde podemos acompanhar as informações enviadas pelo Arduino ao computador. Este monitor serial costuma ser suficiente para grande parte dos casos onde queremos observar os dados de sensores adquiridos pelo Arduino; ele permite que os dados sejam copiados e colados em arquivos de texto ou de planilha para tratamento futuro.

Uma forma de automatizar a coleta de dados da porta serial é usando um *script* em Python. Para isso é necessário ter a biblioteca `pyserial` instalada[16]. Abaixo temos um exemplo de código que lê a porta serial e automaticamente armazena a leitura em um arquivo.

```
import serial


# Lembrar de alterar o nome do arquivo sempre que iniciar um experimento novo
NOME_DO_ARQUIVO = 'resultado.txt'

# Usar aqui o mesmo valor da velocidade de comunicação usada para iniciar a
# comunicação serial no Arduino
BAUD_RATE = 9600

# Alterar o nome da porta serial para a porta onde o Arduino está conectado
PORTA_SERIAL = 'COM3'  # usar esta notação caso esteja no windows
# PORTA_SERIAL = r'/dev/ttyACM0' # usar esta notação caso esteja no linux
```

[16] Disponível em `https://pypi.org/project/pyserial/`.

```
# loop infinito
while True:

    # iniciando comunicação serial
    with serial.Serial(PORTA_SERIAL, BAUD_RATE) as pserial:

        # lendo a porta serial, decodificando e limpando a string
        line = pserial.readline().decode('utf-8').rstrip()

        # exibindo a leitura da porta serial no terminal Python
        print(line)

        # escrevendo resultado em um arquivo
        with open(NOME_DO_ARQUIVO, 'a') as arquivo:
            arquivo.write(line + '\n')
```