Capítulo 7

Portas de entrada/saída no Arduino Mega 2560

Embora o fato dos microcontroladores serem produtos de alta tecnologia, eles acabam ficando sem uso prático se não forem conectados a outros dispositivos adicionais. O aparecimento de tensão nos pinos do microcontrolador não é útil se não for usado para ligar/desligar dispositivos tais como LEDs, relés, displays, etc.

Uma das características mais importantes dos microcontroladores é o número de entradas e saídas que permitem ao projetista conectar dispositivos sensores ou atuadores. O Arduino Mega 2560 tem 54 pinos de entrada/saída digitais de propósito geral, que é uma quantidade mais que suficiente para várias aplicações.

7.1 Introdução

Uma das mais importantes características dos microcontroladores é o número de pinos de entrada/saída, ou, pinos de I/O (Input/Output) de propósito geral (GPIO – *general purpose input/output*). O Arduino Mega 2560 possui 54 pinos de I/O digitais que podem ser individualmente configurados para funcionarem como entradas e/ou saídas, além de 16 pinos de entrada analógica, onde alguns deles também podem ser configurados como portas de I/O digitais.

Os pinos de I/O do Arduino são oriundos de estruturas físicas do microcontrolador Atmega2560 chamadas de Portas. O ATmega 2560 possui 11 portas denominadas A, B, C, D, E, F, G, H, J, K e L. Idealmente, seguindo à sua arquitetura de 8 bits, cada porta deveria possuir 8 pinos de I/O. Entretanto, a porta G possui apenas 6 pinos. No total, o Atmega2560 possui 86 pinos, dos quais apenas 69 são aproveitadas na placa do Arduino Mega 2560.

Por razões práticas, a maioria dos pinos de I/O compartilha de várias funções especiais, que serão exploradas em capítulos posteriores. Se um pino é usado para qualquer uma das funções, ele não pode ser usado para outra. Na verdade, a função de um pino de I/O pode ser alterada enquanto o microcontrolador estiver em funcionamento, mas o hardware externo ao microcontrolador deve estar preparado para se adequar à mudança da nova funcionalidade do pino de I/O.

Quando os pinos do Arduino Mega 2560 estão operando como entradas, a faixa de valores de tensão que pode ser aplicado aos pinos deve ser de, no máximo, -0,5V a 5,5V.

Se os pinos forem usados como entradas digitais, deve-se assegurar que os valores de nível lógico alto e baixo sejam obedecidos. Assim, para o nível lógico baixo, um valor tão próximo de 0V quanto possível deve ser aplicado. Já para o nível alto, um valor tão próximo de 5V deve ser aplicado. Sempre deve ser evitada a aplicação de valores intermediários de tensão em pinos operando como entradas digitais, uma vez que os valores podem não ser distinguidos como nível alto ou baixo pelo hardware do microcontrolador.

7.2 Funções de manipulação de pinos digitais

As funções que fazem a manipulação individual dos pinos no Arduino são basicamente três, duas das quais foram utilizadas nos exemplos do capítulo anterior:

`pinMode(pino, modo)` – Configura um pino como entrada ou saída;
`digitalWrite(pino, valor)` – Escreve um nível lógico em um pino;
`digitalRead(pino)` – Faz a leitura do nível lógico de um pino.

Configurando um pino como entrada ou saída

A função `pinMode(pino, modo)` configura o pino especificado pelo parâmetro **`pino`** como uma entrada ou saída digital.

O parâmetro **`pino`** pode ser uma constante, indicando o número do pino a ser configurado, ou uma variável, cujo valor seja o número do pino.

O parâmetro **`modo`** indica se o pino deve operar como entrada ou saída. Para operar como saída digital, o parâmetro deve ser fornecido como **OUTPUT.** Para operar como entrada digital, o parâmetro deve ser fornecido como **INPUT** ou **INPUT_PULLUP**. As palavras OUTPUT, INPUT e INPUT_PULLUP são palavras reservadas da linguagem Arduino, sendo expressões pré-definidas do compilador. Os pinos do Arduino apresentam a configuração padrão como entradas, portanto, eles não precisam ser explicitamente declarados como entradas.

Os pinos configurados com `pinMode(pino, INPUT)` são ditos estarem em um estado de alta impedância, apresentando uma resistência de entrada de aproximadamente 100 MΩ e fazendo solicitações extremamente pequenas nos circuitos que estão sendo amostrados. Isso significa que é preciso pouca corrente para mudar o pino de entrada de um estado para outro. Isso pode tornar os pinos úteis para tarefas como implementar um sensor de toque capacitivo, ler um fotodiodo ou ler um sensor analógico qualquer. No entanto, isso também significa que os pinos configurados como entrada e que não tenham nada ligado a eles, ou com fios conectados a eles que não estejam conectados a outros circuitos, informarão, durante uma leitura nesses pinos, mudanças aparentemente aleatórias no seu estado lógico, sendo diretamente afetados por ruídos elétricos do ambiente ou acoplamento capacitivo do estado lógico de um pino próximo, o que não é desejado na prática.

É útil garantir que um pino de entrada esteja em um estado lógico conhecido se nenhum sinal de entrada estiver presente. Isso pode ser feito adicionando um resistor de *pullup* (conectado entre o pino e +5V), ou um resistor de *pulldown* (entre o pino e o GND) na entrada. Um resistor de 10 kΩ é um valor comumente utilizado para um resistor de *pullup* ou *pulldown*.

Existem resistores de *pullup*, cujo valor varia entre 20-50 kΩ, incorporados no chip Atmega que podem ser acessados a partir do software. Esses resistores de *pullup* embutidos são acessados definindo o modo da função pinMode( ) como INPUT_PULLUP.

Ao conectar um sensor a um pino configurado com `pinMode(pino, INPUT_PULLUP)`, a outra extremidade deve ser conectada ao GND. No caso de um interruptor simples, por exemplo, isso faz com que a leitura no pino retorne o nível lógico ALTO quando o interruptor estiver aberto e BAIXO quando o interruptor for pressionado, como mostrado na Fig. 1 abaixo, onde um *push-button* normalmente aberto está conectado entre os pinos 2 e GND do Arduino Leonardo.

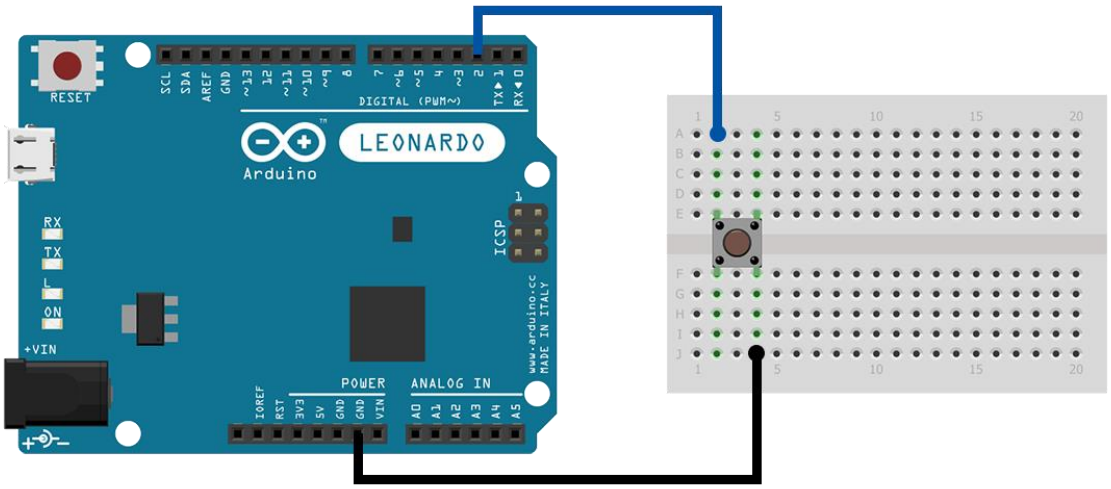

Figura 1 – Push-button normalmente aberto conectado entre os pinos 2 e GND do Arduino Leonardo.

Os resistores de *pullup* são controlados pelos mesmos registradores que controlam o estado lógico de um pino quando este é configurado como saída digital. Consequentemente, um pino que está configurado para que os resistores de *pullup* sejam ativados quando o pino é uma ENTRADA, apresentará nível lógico ALTO se o pino for alternado para uma SAÍDA. Isso também funciona na outra direção, e um pino de saída que é deixado em um estado ALTO terá os resistores *pullup* configurados se mudado para uma ENTRADA.

O pino digital 13 é mais difícil de usar como uma entrada digital do que os outros pinos digitais porque ele possui um LED e um resistor conectados a ele. Se você habilitar o resistor de *pullup* interno de 20k, ele apresentará algo em torno de 1.7 V em vez dos esperados 5 V porque o LED e o resistor em série puxam o nível de tensão para baixo, o que significa que uma leitura nesse pino sempre retorna LOW. Assim, ao usar o pino 13 como uma entrada digital, deve-se definir seu modo como INPUT e usar um resistor *pulldown* externo.

Escrevendo um nível lógico em um pino configurado como saída

A função `digitalWrite(pino, valor)` escreve o nível lógico definido por `valor` no pino especificado pelo parâmetro **`pino`**.

O parâmetro **`pino`** pode ser uma constante indicando o número do pino a ser configurado, ou uma variável cujo valor seja o número do pino.

O parâmetro **`valor`** deve ser **HIGH** ou **LOW**, quando se deseja levar o pino para nível lógico alto ou baixo, respectivamente. Levar o pino para nível lógico alto significa enviar 5 V para o pino especificado. Da mesma forma, levar o pino para nível lógico baixo é enviar 0 V para o pino especificado.

Os pinos configurados como OUTPUT com pinMode( ) são ditos estarem em um estado de baixa impedância. Isso significa que eles podem fornecer uma quantidade substancial de corrente para outros circuitos.

Os pinos do Arduino Mega 2560 podem fornecer ou drenar até 40 mA de corrente. Esta corrente é suficiente para iluminar intensamente um LED, ou controlar alguns sensores, por exemplo, mas não é suficiente para excitar a maioria dos relés, solenoides ou motores.

Um curto circuito em algum pino do Arduino, ou a tentativa de excitar diretamente dispositivos de alta corrente, podem danificar ou destruir os transistores de saída do pino, resultando em um "pino morto" no microcontrolador ou danificando permanentemente todo o chip Atmega 2560.

Lendo o nível lógico em um pino configurado como entrada

A função `digitalRead(pino)` lê o nível lógico presente no pino especificado pelo parâmetro **`pino`**. Se houver um nível lógico alto no pino, a função retorna **HIGH**, e se houver um nível lógico baixo, a função retorna **LOW**.

Normalmente, essa função é utilizada numa atribuição a uma variável, como por exemplo:

```
boolean leitura;
leitura = digitalRead(5);
```

Assim, a variável leitura armazena o estado lógico presente no pino 5. A variável leitura pode então ser usada para verificar, por exemplo, se um botão conectado ao pino foi ou não pressionado.

7.3 Componentes adicionais

Esta seção cobre a maioria dos componentes mais utilizados na prática que se conectam ao microcontrolador, tais como chaves, botões, transistores, LED's, relés, etc.

Chaves e botões

Chaves e botões de pulso (*push-buttons*) são provavelmente os mais simples dispositivos de entrada que possibilitam a detecção do aparecimento de tensão em um pino do microcontrolador. Eles são ligados aos pinos configurados como entradas no microcontrolador. Entretanto, fazer essa conexão não é tão simples quanto parece. A razão para isso é o que se conhece por "*bounce*", ou oscilação, ou ainda, ruído elétrico.

A oscilação nos contatos de uma chave mecânica é um problema comum que deve ser tratado pelo projetista de circuitos com microcontroladores. Esta oscilação causa problemas em alguns circuitos analógicos e digitais, como os microcontroladores, que respondem rapidamente às variações de tensão em seus pinos.

Na figura 2, temos a representação da tensão em um pino de I/O do Arduino, no qual foi conectada uma chave. Quando a chave está em repouso, a tensão no pino de I/O é de 5V. Ao se pressionar a chave, a tensão idealmente deveria cair para zero volt. Em vez disso, ela cai e volta a subir enquanto o chaveamento do contato mecânico da chave não se estabilizar. Isso ocorre rapidamente, durando cerca de 0,01 a até 100 ms, dependendo do tipo de chave e circuito.

Apesar do transitório da tensão ocorrer rapidamente, o microcontrolador percebe todas essas variações de tensão devido à sua alta velocidade de processamento. Assim, por exemplo, ao usar o microcontrolador para contar a quantidade de vezes que uma chave é pressionada, podem ocorrer erros de contagem devido a tais oscilações.

Este problema pode ser facilmente resolvido simplesmente conectando um circuito do tipo filtro RC simples para suprimir essas rápidas mudanças de tensão. Uma vez que o período de duração das oscilações não é definido, os valores dos componentes não são precisamente determinados. Em muitos casos, é recomendado usar os valores mostrados na figura 3.

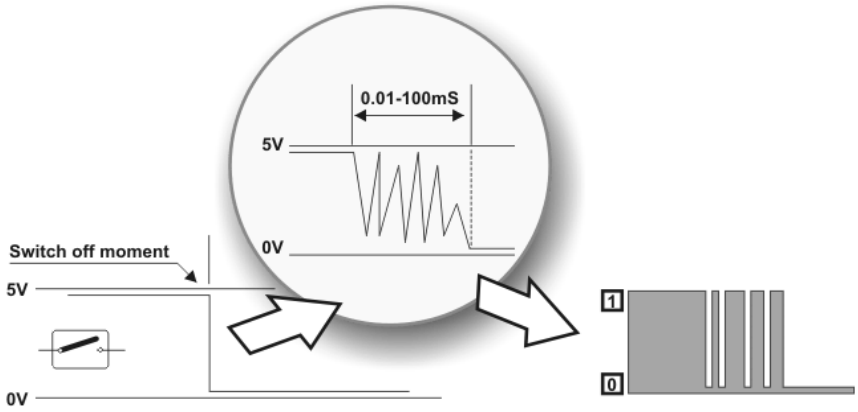

Figura 2 – Representação da tensão em um pino do Arduino onde foi conectado uma chave.

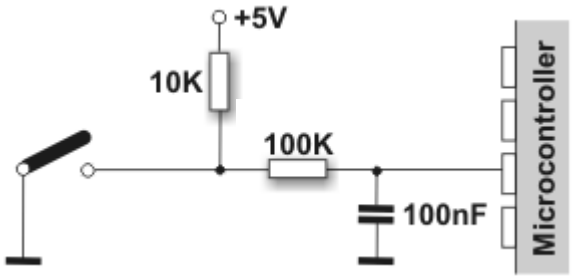

Figura 3 – Filtro RC conectado no pino do microcontrolador para contornar o efeito do chaveamento.

Em adição a esta solução por hardware, há também uma solução simples por software: Quando o programa testa o estado lógico (nível de tensão) de um pino de entrada e detecta uma mudança, o teste deve ser refeito depois de um determinado tempo. Se o programa confirmar a mudança, significa que o botão, ou a chave, realmente mudou de posição e então o programa executa a tarefa associada.

Relés

Um relé (figura 4) é um interruptor elétrico que abre e fecha sob o controle de outro circuito elétrico. É, portanto, ligado aos pinos de saída do microcontrolador e usado para ativar/desativar dispositivos de alta potência, como motores, transformadores, aquecedores, lâmpadas, etc. Existem vários tipos de relés, mas todos eles funcionam da mesma maneira. Quando a corrente flui através de sua bobina, o relé é operado por um eletroímã para abrir ou fechar um ou mais conjuntos de contatos. Similarmente aos acopladores ópticos, não há nenhuma conexão galvânica (contato elétrico) entre os circuitos de entrada e saída.

Figura 4 – Modelos de relés eletromecânicos.

A figura 5 mostra a solução mais comumente empregada para se conectar um relé a um pino do microcontrolador.

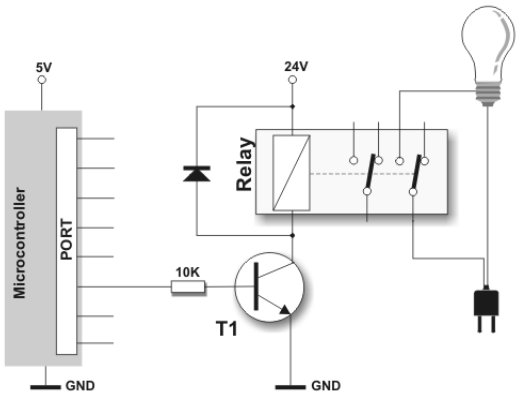

Figura 5 – Ligação de um relé a um microcontrolador.

No circuito mostrado na figura 5, o microcontrolador pode ligar/desligar a lâmpada, que é uma carga considerada de alta potência (quando comparada à potência do próprio microcontrolador) a partir do acionamento do relé. Uma vez que as correntes e tensões fornecidas diretamente nos pinos de saída do microcontrolador são normalmente bem pequenas, isso normalmente é insuficiente para ativar diretamente o relé. Assim, o transistor T1 é utilizado como driver para ativar o relé.

A fim de prevenir o aparecimento de um alto valor de tensão auto induzida, causada por uma interrupção brusca do fluxo de corrente através da bobina quando o relé é desenergizado, um diodo inversamente polarizado, chamado de "diodo de roda livre", é ligado em antiparalelo com a bobina do relé. O objetivo deste diodo é manter a corrente fluindo pela bobina até que o seu valor seja reduzido até zerar completamente, eliminando o aparecimento de picos de tensão no circuito.

7.4 Estruturas condicionais

Chamamos de estrutura condicional as instruções para testar se uma condição é verdadeira ou falsa. Temos basicamente duas estruturas condicionais: o comando **if-else** e o comando **switch-case**.

A estrutura if-else

De uma maneira geral, a estrutura **if** ("se" em português) é utilizado para executar um comando ou bloco de comandos no caso de uma determinada condição ser avaliada como verdadeira. Opcionalmente, é também possível executar outro comando ou bloco de comandos no caso de a condição ser avaliada como falsa.

A forma geral da estrutura **if** é:

```
if(condição) comandoA;
else comandoB;
```

Ou como segue:

```
if(condição)
{
   comandoA1;
   comandoA2;
   ...
}
else
{
   comandoB1;
   comandoB2;
   ...
}
```

em ambos os casos, a cláusula **else** não é obrigatória para o funcionamento da estrutura.

O princípio de funcionamento da estrutura é muito simples: se a condição for verdadeira, será executado apenas o comandoA (ou o bloco de comandos A). Caso a condição seja avaliada como falsa, então será executado apenas o comandoB (ou o bloco de comandos B).

Um detalhe muito importante a ser observado é que nunca, no mesmo teste, os dois comandos ou blocos de comandos da estrutura **if-else** serão executados. Dizemos que eles são mutuamente excludentes.

Observe um exemplo de programa que faz a leitura de duas chaves ligadas a dois pinos do Arduino e usa a estrutura **if** para testar quais teclas foram pressionadas. As chaves são do tipo normalmente abertas ligadas nos pinos 2 e 3 com a outra extremidade de cada chave conectada ao GND. Além disso, dois LEDs são usados para mostrar o estado das duas chaves, um ligado ao pino 12, que acende quando a chave do pino 2 é pressionada, e outro ao pino 13, que acende quando a chave do pino 3 é pressionada.

```
//Entradas e saídas digitais – uso do if-else

void setup()
{
  pinMode(2, INPUT_PULLUP);
  pinMode(3, INPUT_PULLUP);
  pinMode(12, OUTPUT);
  pinMode(13, OUTPUT);
}

void loop()
{
  if(digitalRead(2))
     digitalWrite(12, LOW);
  else
     digitalWrite(12, HIGH);

  if(digitalRead(3))
     digitalWrite(13, LOW);
  else
     digitalWrite(13, HIGH);
}
```

Nesse programa, a função setup( ) configura os pinos 2 e 3 como entradas digitais, além de ativar seus resistores de *pullup* internos. A função também configura os pinos 12 e 13 como saídas digitais.

Na função loop( ), a condição do primeiro comando **if** é a função que faz a leitura digital do pino 2, que pode retornar **HIGH** ou **LOW**. Se a chave estiver aberta, a função de leitura desse pino retorna **HIGH** devido ao resistor de *pullup* interno. O valor **HIGH** é avaliado como verdadeira no **if** e, nesse caso, o comando `digitalWrite(12, LOW)` é executado e o LED conectado a esse pino é desligado. Caso a chave estivesse pressionada, o resultado da leitura digital do pino seria **LOW**, que por sua vez seria avaliado como falso, o que faria com que o comando `digitalWrite(12, HIGH)` fosse executado, ligando o LED conectado ao pino. A mesma lógica ocorre com a chave conectada ao pino 3 e com o LED do pino 13. Dessa forma, com as chamadas da função loop( ), os pinos 2 e 3 são permanentemente lidos e os LEDs dos pinos 12 e 13 são acionados de acordo com o estado das chaves.

A figura 6, na página seguinte, mostra uma montagem onde um Arduino UNO é utilizado para controlar o acionamento de duas cargas (ventilador e lâmpada) conectadas à rede elétrica de 110/220V. Para realizar o chaveamento das cargas, foi utilizado um *shield* com dois relés. Além da alimentação de 5 V, o shield exige do Arduino dois sinais, IN1 e IN2, para poder atuar os relés. Esses sinais polarizam a base de dois transistores que servem de drivers para os relés, semelhante ao que foi apresentado na figura 5. A rede e a carga são conectadas nos terminais comum (C) e normalmente abertos (NO) dos relés, fazendo com que esses funcionem como interruptores eletrônicos, comandados pelo Arduino.

O relé 1 comanda a lâmpada e o relé 2 comanda o ventilador. Os relés são acionados pelos pinos 7 e 8 do Arduino UNO, respectivamente. Foram também usadas duas chaves normalmente abertas, conectadas aos pinos 2 e 3, e dois resistores externos de pulldown, que forçam os pinos ao nível lógico baixo quando as chaves estão em repouso.

Quando a chave da esquerda, conectada ao pino 2, é pressionada, o relé 1 deve ser ativado, fazendo a lâmpada acender. Quando a chave é pressionada novamente, o relé 1 deve ser desativado, desligando a lâmpada. A chave da direita, por sua vez, apresenta um comportamento semelhante, ativando/desativando o relé 2 quando pressionada, fazendo com que o ventilador seja ligado ou desligado.

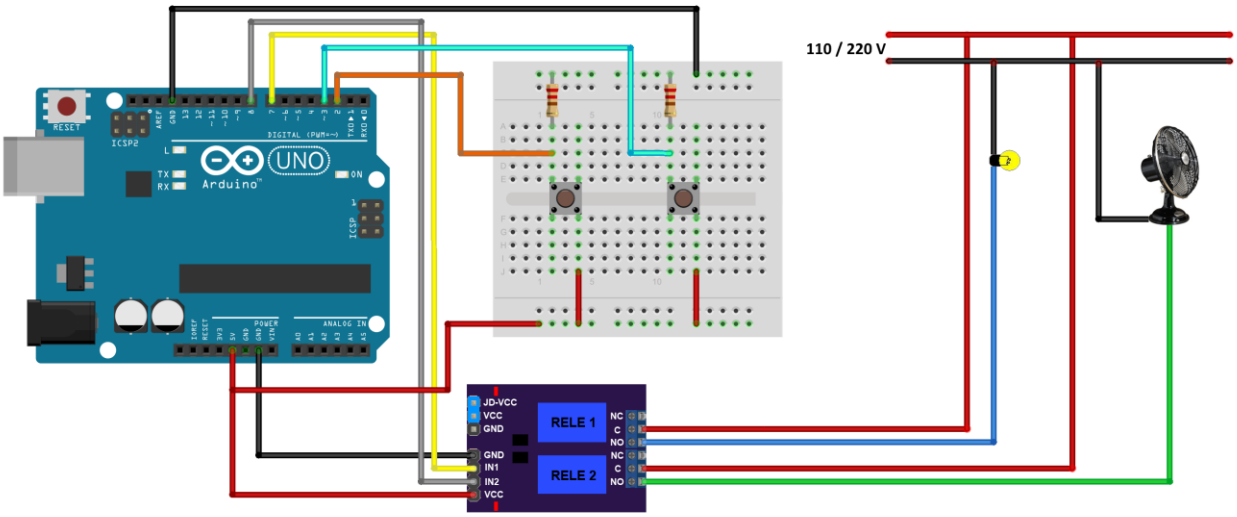

Figura 6 – Arduino UNO controlando duas cargas conectadas à rede elétrica.

O programa a seguir implementa o controle descrito anteriormente, além de introduzir novos conceitos da linguagem Arduino.

```
//Entradas e saídas digitais – controle de cargas

#define  botao1         2
#define  botao2         3
#define  lampada        7
#define  ventilador     8

void setup()
{
  pinMode(bota1, INPUT);
  pinMode(botao2, INPUT);
  pinMode(lampada, OUTPUT);
  pinMode(ventilador, OUTPUT);

  digitalWrite(lampada, LOW);
  digitalWrite(ventilador, LOW);
}

void loop()
{
  if(digitalRead(botao1))
  {
     digitalWrite(lampada, !digitalRead(lampada));
     delay(500);
  }

  if(digitalRead(botao2))
  {
     digitalWrite(ventilador, !digitalRead(ventilador));
     delay(500);
  }
}
```

No programa acima, as linhas que iniciam com #define são diretivas do compilador e não precisam de ponto e vírgula ao final das declarações.

A diretiva #define é um componente útil que permite que o programador atribua um nome a um valor constante, antes que o programa seja compilado. Constantes definidas no Arduino dessa forma não ocupam nenhum espaço de memória de programa no microcontrolador. O compilador substituirá as referências a essas constantes pelo valor definido, em tempo de compilação.

Isso pode ter alguns efeitos colaterais indesejados se, por exemplo, uma constante que tenha sido definida pelo uso dessa diretiva for incluída em alguma outra constante ou nome de variável. Nesse caso, o texto seria substituído pelo valor definido (ou texto definido).

Dessa forma, por exemplo, a declaração "#define botao1 2" atribui o valor 2 ao nome botao1. Sempre que o compilador encontrar o texto definido, ele será substituído pelo valor 2. Isso é útil quando o programador deseja substituir um valor em diferentes partes do texto do programa. Para que não seja necessário alterar as várias partes, basta alterar o valor na declaração #define que o compilador, em tempo de compilação, substituirá todas as ocorrências daquele nome pelo novo valor.

No programa, a função setup( ) inicialmente configura os pinos de entrada e saída. Agora, os comandos pinMode utilizam os nomes que foram definidos pelas diretivas para referenciar os números dos pinos que serão usados no programa. Posteriormente, a função setup( ) inicia os pinos de saída com nível lógico baixo, para iniciar com os relés da lâmpada e ventilador desativados.

Na função loop ( ), as estruturas **if** monitoram quando os botões são pressionados. A condição do primeiro **if** é o retorno da leitura do botão que está conectado ao pino 2 (botao1). Se o botão não estiver pressionado, essa leitura retorna **LOW** e a condição é avaliada como falsa, o que faz com que o bloco de código não seja executado. Se o botão for pressionado, a leitura retorna **HIGH** e a condição é avaliada como verdadeira. Assim, na execução do bloco de código, o microcontrolador leva o pino 7 (definido pela palavra lampada) para o nível lógico definido pela expressão !digitalRead(lampada), que é a negação do retorno da leitura ao próprio pino 7. Ou seja, se o pino estava em nível lógico baixo, ele passará a nível lógico alto, e vice-versa. Assim, cada vez que o botão é pressionado, o estado lógico do pino 7 é invertido, ligando ou desligando a carga conectada ao relé 1.

O comando delay(500) tem a função de aguardar meio segundo após o acionamento do relé para permitir que o botão seja solto sem que haja um novo acionamento do relé. Caso esse comando não existisse, a estrutura **if** seria encerrada logo após o acionamento do relé e, com a nova chamada da função loop( ), o botão ainda estaria pressionado, o que levaria a modificar, mais uma vez, o estado do relé. Assim, o estado do relé poderia mudar diversas vezes enquanto o botão estivesse pressionado.

Funcionamento semelhante ocorre com a segunda estrutura **if**, responsável pelo monitoramento do segundo botão e pelo acionamento do segundo relé.

A estrutura *switch-case*

Em alguns casos, como na comparação de uma determinada variável com diversos valores diferentes, a estrutura **if** pode torna-se um pouco confusa, ou pouco eficiente. Para essas situações, o uso da estrutura **switch** se mostra mais apropriado. A estrutura **switch** permite a realização de comparações sucessivas, de uma forma muito mais elegante, clara e eficiente que a estrutura **if**. Vejamos então o formato geral da estrutura:

```
switch(variável)
{
   case constante1:
      comandoA;
      ...
      break;
   case constante2:
      comandoB;
      ...
      break;
   case constante3:
      comandoC;
      ...
      break;
   default:
      comandoZ;
      ...
      break;
}
```

O valor da variável é testado contra as constantes especificadas pelas cláusulas *case*. Caso a variável e a constante possuam o mesmo valor, então os comandos seguintes àquela cláusula *case* serão executados até a cláusula *break*, que encerra a estrutura **switch**, o que significa dizer que as demais cláusulas *case* não serão testadas.

Caso o valor não encontre correspondentes nas constantes especificadas pelas *cases*, então os comandos especificados pela cláusula *default* são executados.

Repare que cada sequência de comandos da cláusula *case* é encerrada por uma cláusula *break*. Caso esta cláusula seja omitida, todos os comandos subsequentes ao *case* cujo valor da variável foi coincidente serão executados até que seja encontrada outra cláusula *break*, ou ao atingir o final da estrutura **switch**.

O código seguinte exemplifica o uso da estrutura **switch**. Para esse programa, há uma chave normalmente aberta conectada ao pino 2 e quatro LEDs conectados aos pinos 10, 11, 12 e 13. Ao pressionar a chave, um dos quatro LEDs será aleatoriamente aceso e permanecerá assim enquanto a chave estiver pressionada, de acordo com a lógica descrita em seguida.

```
//Entradas e saídas digitais – Uso do switch-case

unsigned int contador=0;

void setup()
{
  pinMode(2, INPUT_PULLUP);
  pinMode(10, OUTPUT);
  pinMode(11, OUTPUT);
  pinMode(12, OUTPUT);
  pinMode(13, OUTPUT);
}

void loop()
{
  if(!digitalRead(2))
  {
    switch(contador)
    {
       case 0:
         digitalWrite(10, HIGH);
         break;
       case 1:
         digitalWrite(11, HIGH);
         break;
       case 2:
         digitalWrite(12, HIGH);
         break;
       case 3:
         digitalWrite(13, HIGH);
         break;
    }
    while(!digitalRead(2));
  }
  else
  {
    digitalWrite(10, LOW);
    digitalWrite(11, LOW);
    digitalWrite(12, LOW);
    digitalWrite(13, LOW);
  }

  ++contador;
  if(contador==4)
    contador=0;
}
```

A linha `unsigned int contador=0;` declara a variável global contador e atribui a ela o valor 0.

A função setup( ) simplesmente configura os pinos de entrada e saída.

Na função loop( ), inicialmente, podemos identificar uma estrutura **if** que testa se a chave no pino 2 está pressionada. Perceba que a condição de teste é a negação (`!`) da leitura digital no pino 2, isto é, se a leitura no pino 2 retorna **HIGH** (no caso em que a chave não estiver pressionada), a condição é avaliada como falsa, e se a leitura retorna **LOW** (no caso da chave estar pressionada), a condição é avaliada como verdadeira.

Dessa forma, o bloco de comandos da estrutura **if** é executado se a chave estiver pressionada. Caso contrário, ou seja, se a chave não estiver pressionada, o bloco de comandos do **else** é executado e, nesse caso, o bloco de comandos faz com que todos os LEDs sejam desligados, como é esperado, e permanecem assim enquanto a chave estiver em repouso.

Após a estrutura **if-else** há o incremento da variável contador. Quando essa variável atinge o valor 4, a estrutura **if**, logo após o comando de incremento, faz com que a variável retorne ao valor 0. Dessa forma, a cada nova chamada da função loop( ), a variável contador vai mudando seu valor de 0 para 1, de 1 para 2, e assim sucessivamente até atingir o valor 4, quando retorna ao valor 0.

Voltando à estrutura **if** inicial, quando a chave é pressionada, a estrutura **switch** é executada. Essa estrutura testa a variável contador contra os valores 0, 1, 2 e 3, ativando um dos LEDs dos pinos 10, 11, 12 ou 13, conforme seja o valor correspondente da variável contador. Como o valor da variável depende do momento em que a chave é pressionada, não há como prever qual LED será aceso.

Após a estrutura **switch**, há a linha `while(!digitalRead(2));` que faz com que o programa fique aguardando, sem executar qualquer comando, enquanto a chave estiver pressionada, o que mantém aceso o LED previamente ativado. Para entender porque isso ocorre, basta analisarmos o funcionamento do laço **while**: A condição do laço **while** é a negação (`!`) da leitura digital do pino 2. Se a chave conectada ao pino estiver pressionada, a condição é avaliada como verdadeira e o bloco de comandos associado ao **while** é executado enquanto essa condição permanecer. Entretanto, perceba que não há nenhum bloco de comandos referente a esse laço. Ao invés disso, há simplesmente um ponto e vírgula ";" após o **while**, o que indica que nada deve ser executado enquanto a chave estiver pressionada. Quando a chave é solta, a condição é avaliada como falsa e o programa segue o fluxo normal de execução.

7.5 Registradores das portas

Os registradores das portas permitem uma manipulação de mais baixo nível nos pinos do microcontrolador das placas Arduino, garantindo operações mais rápidas do que usando as funções internas digitalWrite( ) ou digitalRead( ). Como visto na introdução desse capítulo, o microcontrolador Atmega 2560 possui 11 portas cujos pinos digitais, correspondentes aos 54 pinos digitais da placa Arduino Mega 2560, são os seguintes:

Pino na placa Arduino	Porta no microcontrolador	Pino no microcontrolador
0	E	PE0
1	E	PE1
2	E	PE4
3	E	PE5
4	G	PG5
5	E	PE3
6	H	PH3
7	H	PH4
8	H	PH5
9	H	PH6
10	B	PB4
11	B	PB5
12	B	PB6
13	B	PB7
22-29	A	PA0-PA7
30-37	C	PC7-PC0
38	D	PD7
39-41	G	PG2-PG0
42-49	L	PL7-PL0
50-53	B	PB3-PB0

Cada porta é controlada por três registradores (**DDRx**, **PORTx** e **PINx**, o **x** represetan a letra da porta específica), que também são variáveis definidas na linguagem arduino. O registrador **DDRx** configura a direção do pino, como entrada ou saída. O registrador **PORTx** especifica o estado lógico do pino quando configurado como saída digital e o registrador **PINx** fornece o estado dos pinos quando configurados como entrada. Os registradores **DDRx** e **PORTx** são de escrita e leitura. Os registradores **PINx** correspondem ao estado das entradas e apenas são de leitura.

Cada bit desses registradores corresponde a um único pino. Por exemplo, o bit mais significativo de **DDRB**, **PORTB** e **PINB** referem-se ao pino PB7 (pino digital 13). Lembre-se que alguns bits de uma porta podem ser usados para outras finalidades que serão exploradas em capítulos posteriores. Portanto, deve-se ter cuidado para não alterar os valores dos bits de registradores de uso especial correspondentes a esses pinos. Note-se ainda que os pinos 0 e 1 são usados para comunicação serial para programação e depuração do Arduino, portanto, a mudança desses pinos geralmente deve ser evitada, a menos que seja necessário para funções de entrada ou saída seriais. Esteja ciente de que isso pode interferir no download ou depuração do programa.

Para configurar um pino como entrada, devemos resetar o bit correspondente àquele pino no registrador **DDRx** equivalente, e para configurá-lo como saída, devemos setar o bit. Por exemplo:

```
DDRC = 0b11111110;
```

configura o pino PC0 (pino 37) como entrada e os pinos PC1-PC7 (pinos 36-30) como saídas. Outro exemplo:

```
DDRC = DDRC | 0b11110000;
```

configura os pinos PC4-PC7 como saídas sem alterar a direção dos pinos PC0-PC3.

Para habilitar os resistores de *pullup* de um bit, basta configurá-lo como entrada e em seguida setar o bit correspondente no registrador **PORTx**.

Para levar um pino configurado como saída para o nível lógico alto ou baixo, setamos ou resetamos, respectivamente, o bit correspondente àquele pino no registrador **PORTx** equivalente. Por exemplo:

```
PORTC = 0b10101000;
```

força os pinos 30, 32 e 34 ao nível lógico alto se eles tiverem sido configurados como saídas.

Ao fazer uma leitura no registrador **PINx**, os bits setados correspondem a níveis lógicos altos presentes nos pinos correspondentes, e vice-versa. O registrador retorna a leitura de todos os pinos da porta ao mesmo tempo. Embora o registrador **PINx** seja apenas de leitura, escrever uma lógica 1 em algum de seus bits resulta na alternancia do estado lógico do bit correspondente no registrador **PORTx** e, consequentemente, no pino correspondente, caso esteja configurado como saída.

De um modo geral, manipular os pinos a partir do acesso direto aos registradores tem algumas implicações negativas:

* Se torna muito mais difícil para depurar e manter o código, e é menos claro para outras pessoas entenderem. Só leva alguns microssegundos para o processador executar o código, mas pode levar horas para que se descubra por que ele não está funcionando corretamente e consertá-lo! Normalmente, é muito melhor escrever um código de maneira mais óbvia.
* O código é menos portável. Se você usa digitalRead( ) e digitalWrite( ), é muito mais fácil escrever um código que pode ser executado em qualquer placa Arduino, uma vez que os registradores de controle das portas podem ser diferentes em cada tipo de microcontrolador.
* É muito mais fácil causar mal funcionamento no microcontrolador com o acesso direto à porta. Seria muito fácil, por exemplo, acidentalmente fazer com que uma porta serial parasse de funcionar alterando a direção dos pinos de entrada e saída serial.

Entretanto, as implicações positivas na manipulação direta de portas são:

* Talvez seja necessário ativar e desativar os pinos de modo muito mais rápido, em frações de microssegundo. As funções digitalRead( ) e digitalWrite( ) são formadas por uma dúzia de linhas de código, que são compiladas em algumas instruções de máquina. Cada instrução de máquina requer um ciclo de clock a 16 MHz, que pode se somar em aplicações sensíveis ao tempo, como aplicações de processamento de sinais em tempo real. O acesso direto à porta pode fazer o mesmo trabalho em muito menos ciclos de clock.
* Às vezes é necessário acionar vários pinos de saída exatamente ao mesmo tempo. Por exemplo, chamando `digitalWrite(10, HIGH);` seguido de `digitalWrite (11, HIGH);` fará com que o pino 10 vá a nível lógico alto alguns microssegundos antes do pino 11, o que pode confundir certos circuitos digitais externos sensíveis ao tempo. Alternativamente, pode-se levar ambos os pinos ao nível lógico alto exatamente no mesmo instante usando o registrador `PORTB = PORTB | 0b00110000;`
* Se há restrições quanto ao espaço de memória de programa disponível, pode-se usar a manipulação direta de portas para reduzir o código. Exige-se muito menos bytes de código compilado para configurar vários pinos de hardware simultaneamente através dos registradores de portas do que para configurar cada pino separadamente. ■