INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SERGIPE

COORDENADORIA DE ELETRÔNICA

SISTEMAS PROGRAMÁVEIS

Guia de Estudos

Prof. Fábio Prudente

Aracaju, 2019

Índice:

Introdução

Nesta disciplina, aprenderemos a programar sistemas baseados em microcontroladores (no caso, nossa plataforma será o Arduino). Tendo sempre em mente os quatro requisitos abaixo, o objetivo dessa disciplina é criar sistemas que sejam:

Dedicados: ao contrário dos sistemas de propósito geral, como os smartphones e PCs, que podem abrir e fechar diferentes aplicativos sob demanda do usuário, os sistemas dedicados cumprem um objetivo específico: exemplos: o controle de um forno de microondas, máquina de lavar, injeção eletrônica de combustível… na prática, esse requisito implica que sistemas dedicados executam o mesmo programa, eternamente. No Arduino, essa característica é encontrada na função `loop()`.

Reativos: Nosso interesse, como estudantes de eletrônica, é desenvolver sistemas capazes de interagir com o mundo físico, através de sensores e atuadores. Um Arduino pode, por exemplo, ler o valor de um sensor de temperatura, e acionar o compressor de um condicionador de ar… ou processar conjuntamente as informações de diversos sensores de localização (GPS, ultrassom, óticos…) e comandar o acionamento de motores para mover um robô autônomo.

Tempo-Real: Além de interagir com o mundo físico, desejamos que essa reação considere a medida precisa do tempo. Desde a simples temporização em um forno de microondas, ao cálculo dinâmico de estabilidade de um drone, a medição do tempo, é fundamental.

Multi-Tarefa: A maioria das aplicações requer que o sistema seja capaz de processar várias informações “ao mesmo tempo”. Um robô, um drone, um sistema residencial de alarme, ou um sistema de controle industrial… todos tem que processar informações de vários sensores, comunicar com outros sistemas, gerar sinais para diversos atuadores… tudo isso “ao mesmo tempo”. O Arduino tem apenas uma CPU, capaz de executar uma única instrução por vez… mas ele é capaz de executar 16 milhões de instruções a cada segundo. Então podemos criar várias pequenas funções (ou “tarefas”) de forma que o arduino possa executar todas elas, sequencialmente, em tempos muito pequenos, criando assim a ilusão de que ele está executando todas “ao mesmo tempo”.

Para construir os 4 conceitos acima, essa disciplina precisará trabalhar em três eixos, que serão desenvolvidos em paralelo:

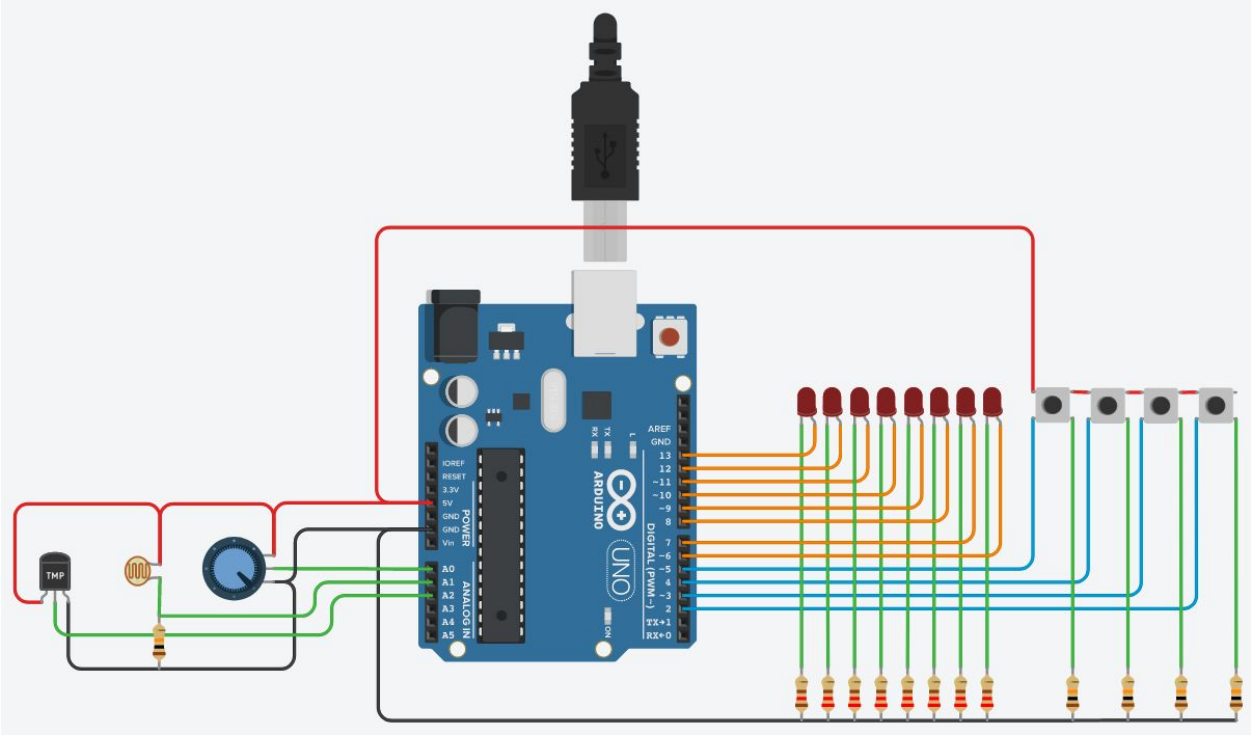
Hardware: como as interfaces nativas do Arduino vão interagir com o mundo externo: circuitos analógicos, digitais, sensores, atuadores, módulos, sistemas, etc…

Lógica de Programação: como vamos organizar o nosso raciocínio, de forma a imaginar os algoritmos que irão resolver nossos problemas.

Linguagem de Programação: como os algoritmos serão descritos em uma linguagem específica de programação (nesse caso, a linguagem C), utilizando a biblioteca padrão de funções do Arduino, ou outras bibliotecas extras, dependendo da aplicação.

Simulador

O projeto abaixo apresenta o Arduino conectado a vários botões, LEDs (dispositivos digitais) e a um potenciômetro, um LDR (sensor de luz) e um sensor de temperatura (dispositivos analógicos). Use esse hardware para todos os exercícios a seguir.



(clique na imagem para abrir o projeto no simulador)

Comunicação Serial

Transmitindo dados

Conceitos:

- Estrutura básica de um "sketch" Arduino: funções setup() e loop()
- Declaração de Variáveis / Tipos de Dados (byte e char) / qualificador static
- Operadores: Atribuição (=) / incremento (++)
- função delay()
- Comunicação Serial: métodos begin(), print(), println() Execute o programa acima, e observe a saída no Monitor Serial:

```
void setup() {
  Serial.begin(9600);     // inicializa comunicação serial (9600 bits/s)
}

void loop() {
  delay (100);            // pausa o programa, por 100ms
  static byte n = 0;      // declara variável n, do tipo byte, e inicializa com 0
  char c = n;             // declara variável c, do tipo char, e atribui valor de n
  Serial.print ( n );     // imprime o valor como um número (pq???)
  Serial.print ( " | " );  // imprime uma string ( " | " )
  Serial.println ( c );    // imprime o mesmo valor como um caractere (pq???)
  n++;                    // incrementa valor de n (faz n = n + 1)
}
```

Exercício:

1. Qual a diferença entre os métodos `Serial.print()` e `Serial.println()`?
2. A sentença `char c = n;` faz com que o valor de n seja atribuído a c, ou seja, a partir desse ponto, as duas variáveis terão exatamente o mesmo valor. Sendo assim, por que essas variáveis são impressas de formas diferentes no monitor serial?
3. O programa acima imprime uma tabela que relaciona o código correspondente a cada caractere. Essa tabela é conhecida como a tabela ASCII. Pesquise sobre a tabela ASCII, e verifique:
    a. Por que os caracteres de 0 a 31 não são impressos?
    b. Qual o código correspondente ao 'A'?
    c. Qual o código correspondente ao 'a'?
    d. Qual o código correspondente ao '0' (dígito zero)?
    e. Qual o código correspondente ao ' ' (espaço)?
4. Por que a variável n foi declarada como **`static`**? o que acontece no programa se esse qualificador for omitido? Explique.
5. Por que a contagem de n passa automaticamente de 255 para 0?
6. Modifique a declaração de n para: **`static int n = 0;`** o que acontece com a contagem?
7. Modifique o programa para que ele imprima o valor de **`n`** nos formatos: decimal, hexadecimal e binário, e depois o valor de **`c`**.

Recebendo dados

Conceitos:

- Comunicação Serial: métodos available(), read()
- Estruturas de Controle: if / else

```
void setup() {
  Serial.begin(9600);  // inicializa comunicação serial (9600 bits/s)
}

void loop() {
  delay (10);          // (somente para a simulação)
  byte n = 0;          // declara variável n, e inicializa com valor 0
  char c;              // declara variável c (não inicializa)

  if ( Serial.available() > 0 ) { // se existem caracteres na fila de entrada
    c = Serial.read();            //   leia 1 caractere da fila de entrada
    n = c;                        //   atribua a n o valor de c
    Serial.print ( n );           //   imprime o valor como um número
    Serial.print ( " | " );       //   imprime a string " | "
    Serial.println ( c );         //   imprime o valor como um caractere
  }
}
```

Execute o programa acima, DIGITE algo no Monitor Serial e observe a saída:

Exercício:

1. Para cada caractere enviado ao Arduino, esse programa faz com que o Arduino imprima de volta o código correspondente (e novamente o caractere).
    a. Qual o código correspondente ao ‘{‘?
    b. Quais os códigos correspondentes ao ‘a’? e ao ‘á’? e ao ‘à’?
2. Digite uma palavra (seu nome, por exemplo). Observe que programa processará um caractere de cada vez. Explique.
3. Nesse programa, nenhuma variável foi declarada como static. Por quê?
4. Leia sobre os Operadores de Comparação, e acrescente a esse programa as seguintes funcionalidades:
    a. se o caractere recebido estiver entre ‘0’ e ‘9’, o Arduino deverá imprimir a mensagem: “Categoria: Dígito numérico”;
    b. se o caractere recebido estiver entre ‘A’ e ‘Z’, o Arduino deverá imprimir a mensagem: “Categoria: Letra maiúscula”;
    c. se o caractere recebido estiver entre ‘a’ e ‘z’, o Arduino deverá imprimir a mensagem: “Categoria: Letra minúscula”;
    d. para qualquer outro caractere, imprimir a mensagem “Categoria: outros”
5. Agora que você já domina a estrutura de controle if / else, volte ao exemplo anterior (Transmitindo dados) e modifique o programa, para que ele imprima somente os caracteres de 0 a 127 (após 127, volta a 0...).

Entradas e Saídas Digitais:

Conceitos:

- Variáveis Locais e Globais (escopo)
- Declaração de constantes (const):
- Entradas e Saídas Digitais: funções pinMode(), digitalRead() e digitalWrite()

```
// Declarações Globais
const byte LED1   = 13;          // declara constante LED1   = 13
const byte BOTAO1 =  5;          // declara constante BOTAO1 =  5

void setup() {
  pinMode ( LED1, OUTPUT );      // configura pino 13 como SAÍDA
  pinMode ( BOTAO1, INPUT );     // configura pino  5 como ENTRADA
  Serial.begin( 9600 );          // inicializa comunicação serial a 9600 bits/s
}

void loop() {
  delay (10);                    // (somente para a simulação)
  bool bt = digitalRead(BOTAO1); // lê BOTAO
  Serial.print( "botao = ");     // imprime texto entre aspas
  Serial.println( bt );          // imprime valor de bt
}
```

Execute o programa, pressione o botão no pino 5, e verifique o que acontece no Monitor Serial

Exercício:

1. Qual a diferença entre variáveis locais e globais? quando devemos usar cada uma?
2. Considere somente 1 LED (pino 13), e 1 botão (pino 5).
    a. Faça com que o LED acenda quando o botão for pressionado, e apague, caso contrário.
    b. Faça com que o LED apague quando o botão for pressionado, e acenda, caso contrário.
    c. Faça com que o LED pisque, com período 1s, quando o botão for pressionado, e apague, caso contrário.
3. Considere somente 1 LED (pino 13), e 2 botões (pinos 5 e 4):
   DICA: use Operadores Booleanos (&& e ||)
    a. Faça com que o LED acenda quando o botão 1 for pressionado, e apague, somente quando o botão 2 for pressionado
    b. Faça com que o LED acenda quando os dois botões estiverem pressionados ao mesmo tempo, e apague somente quando os dois botões estiverem liberados simultaneamente.
    c. Faça com que o LED acenda quando os dois botões estiverem pressionados ao mesmo tempo, e apague em qualquer outra situação.
4. Faça com que o LED acenda ao receber o caractere ‘A’ da Serial, e apague quando receber o caractere ‘a’.
5. Faça com que o LED acenda ao receber qualquer letra maiúscula, e apague ao receber qualquer letra minúscula.

Entradas Analógicas e Saídas PWM

Entradas Analógicas

Conceitos:

- Leitura do conversor Analógico / Digital: analogRead()

Considere o código abaixo:

```
const byte POT = A0;      // Potenciômetro conectado à entrada Analógica 0
                          // OBS: Entradas Analógicas não precisam do pinMode
void setup() {
  Serial.begin(9600);     // inicializa comunicação serial
}

void loop() {
  delay (10);                       // somente para a simulação
  int valor_pot = analogRead(POT); // lê valor analógico do potenciômetro
  Serial.println( valor_pot );     // imprime valor lido na Serial
}
```

Exercício:

1. Execute esse código, gire o potenciômetro, e observe o resultado no monitor Serial: Qual o valor máximo? Qual o valor mínimo?
2. Considere 1 LED no pino 13. Faça com que esse LED acenda quando o valor do potenciômetro for maior que 50%, e apague caso contrário.
3. Considere agora 3 LEDs: faça com que cada LED acenda de acordo com a tabela abaixo:

valor do potenciômetro	LED aceso
0 a 341	1
342 a 682	2
683 a 1023	3

4. Considere 1 LED e duas entradas analógicas (potenciômetro e LDR). Faça com que o LED acenda quando o valor do LDR for menor que o valor do potenciômetro.
5. Pesquise sobre o LM35 (sensor de temperatura).
    a. Implemente código para imprimir na Serial o valor lido no pino A2 (sensor LM35). Ajuste o sensor, no simulador, para dois valores quaisquer de temperatura, e anote os valores correspondentes na leitura do pino A2. Por exemplo: ajuste para 0ºC e para 100ºC… ou ajuste para -40ºC (mínimo do simulador) e +125ºC (máximo do simulador)
    b. Leia sobre a função `map()`, e use-a para converter o valor lido no pino analógico para o valor correspondente em ºC, de acordo com suas medições feitas no item anterior.
    c. Perceba que a função map() trabalha somente com números inteiros e, com isso, a aproximação da temperatura fica um pouco imprecisa. Usando os mesmos dados obtidos no item (a), deduza você mesmo uma expressão para converter o valor lido no valor correspondente em ºC. DICAS: Estude sobre regra de 3 (sim, aquela da matemática), e use o tipo de dado `float` para o resultado.
    d. Pesquise sobre como converter a temperatura de ºC para ºF, e implemente código para imprimir na Serial o valor da temperatura, em ºC e ºF.
    e. Faça com que o LED acenda somente quando a temperatura for maior que 60ºC

Saídas PWM

Conceitos:

- Os pinos marcados com o símbolo '~', na placa do Arduino, possuem a função PWM. Essa funcionalidade é acessada através da função `analogWrite()`.

```
const byte LED1 = 12;       // pino 12 não possui a função PWM
const byte LED2 = 11;       // pino 11 pode ser usado como PWM

void setup() {
  pinMode ( LED1, OUTPUT ); // configura o pinos 11 e 12 como SAÍDAs
  pinMode ( LED2, OUTPUT );
  digitalWrite ( LED1, 1 ); // acende LED no pino 12, para comparar o brilho
  Serial.begin ( 9600 );
}

void loop() {
  delay (50);               // pausa a execução por 50ms
  static byte n = 0;        // o valor de n controlará o brilho do LED...
  analogWrite( LED2, n );   // configura duty-cycle = n
  Serial.println ( n );     // imprime o valor de n
  n++;                      // incrementa n
}
```

O programa acima fará com que o brilho do LED conectado ao pino 11 aumente gradativamente. O LED no pino 12 foi aceso somente para que você possa comparar visualmente o brilho dos dois…

Exercício:

1. No simulador, conecte um osciloscópio para visualizar o sinal no pino 11, e observe a relação entre a forma de onda no osciloscópio, o brilho do LED e o valor de n, no monitor Serial.
2. Troque as declarações de LED1 e LED2, e observe novamente o comportamento. Anote suas conclusões, e volte o programa para o original.
3. Considere agora o Arduino conectado a 1 LED, no pino 11, e a dois botões, nos pinos 5 e 4:
    a. Faça com que um dos botões aumente o brilho do LED, e o outro diminua.
    b. cuide para que o acionamento dos botões não ultrapassem os limites (0 a 255)
4. Além do pino 11, quais outros pinos do Arduino Uno possuem a função PWM?
    a. Controle o brilho de outro LED, usando agora a leitura do potenciômetro
    b. A função **`analogRead()`** retorna valores de 0 a 1023, mas a função **`analogWrite()`** recebe valores de 0 a 255. Compatibilize as duas, para que 100% no potenciômetro corresponda a 100% no LED.

Um simples pulso num botão…

Conceitos:

- Vamos usar apenas os conceitos que já vimos acima, para resolver um problema (aparentemente) simples: detectar um pulso em um botão (ou em qualquer sinal de entrada)
- O objetivo, aqui, é compreender a importância de pensarmos no Algoritmo, antes de escrevermos o Programa.

Considere o Arduino conectado somente a 1 LED e 1 botão. Deseja-se um comportamento muito simples: a cada pulso no botão, o LED deverá inverter seu estado, ou seja: desejamos fazer a função de um botão liga/desliga, presente em qualquer equipamento eletrônico (um pulso no botão, liga, outro pulso no botão, desliga).

Vamos nos habituar a pensar no Algoritmo (uma sequência lógica de passos para resolver o problema, descritos em uma linguagem informal), antes de pensarmos no código (sequência de instruções, escritas em uma linguagem formal de programação):

Solução 1: uma primeira tentativa...

O algoritmo para solucionar esse problema, seria:

> *leia o estado do botão*
> *se botão estiver pressionado,*
> *inverta o estado do LED*

Tendo o algoritmo em mente, já podemos escrever o programa

```
// Declarações Globais
const byte LED1   = 13;          // declara constante LED1   = 13
const byte BOTAO1 =  5;          // declara constante BOTAO1 =  5

void setup() {
  pinMode ( LED1, OUTPUT );      // configura pino 13 como SAÍDA
  pinMode ( BOTAO1, INPUT );     // configura pino  5 como ENTRADA
  Serial.begin( 9600 );          // inicializa comunicação serial a 9600 bits/s
}

void loop() {
  delay (500);                   // <- pausa bem grande para vermos o que acontece
                                  // o algoritmo começa aqui:
  bool bt = digitalRead(BOTAO1);  // lê BOTAO1
  if ( bt == 1 ) {               // se o botão está pressionado,
                                 //   inverta o LED1
    digitalWrite (LED1, ! digitalRead(LED1));
  }
}
```

Execute o programa acima, e veja que ele não faz exatamente o que esperávamos… se o botão ficar pressionado, observa-se que o LED ficará piscando, e não era isso que nós queríamos!

De fato, o algoritmo que pensamos, acima, não está correto. Como tudo no Arduino será executado repetidamente, dentro da função **`loop()`**, o que vai acontecer é que, a cada nova execução, se o botão estiver pressionado, o LED será novamente invertido… e isso acontecerá numa velocidade muito alta! - aqui está lento, somente por causa do **`delay(500)`**.

Portanto, temos que melhorar o nosso algoritmo, fazendo com que ele **memorize** se já fez a inversão do LED para aquele pulso, para que, numa nova execução, a ação de inverter o LED não seja repetida. Sempre que precisarmos **memorizar** algo, significa que usaremos uma **variável** para isso.

Solução 2: usando "Flag", para sinalizar se uma operação já foi feita

Essa solução usa o conceito de "*Flag*" ("bandeira"), que é simplesmente o uso de uma variável para sinalizar que a ação de inverter um determinado LED já foi executada e, portanto, não deve ser repetida, até que o botão correspondente seja liberado.

Vamos ao Algoritmo:

leia o estado do botão
se o botão está pressionado,
    se ação ainda não foi executada,
        inverta o LED
        memorize que a ação já foi executada (flag)
caso contrário (botão não pressionado),
    libere a flag, para o próximo pulso do botão

Código: somente a função loop()

```
void loop() {
  delay (10);                     // somente para o simulador
  static bool acao_executada = 0; // flag

                                  // o algoritmo começa aqui:
  bool bt = digitalRead(BOTAO1);  // leia o estado do botão
  if ( bt == 1 ) {                // se o botão está pressionado,
    if ( acao_executada == 0 ) {  //   se ação ainda não foi executada,
                                  //     inverta o LED
      digitalWrite (LED1, ! digitalRead(LED1));
      acao_executada = 1;         //     memorize que a ação já foi executada
    }
  }
  else {                          //   caso contrário (botão liberado),
    acao_executada = 0;           //     libere a flag, para o próximo pulso
  }
}
```

Execute o código acima, e verifique o comportamento. Funciona!!!

Verifique atentamente como o algoritmo foi convertido em programa.

Solução 3: usando conceito de "valor anterior", para avaliar a evolução de um SINAL

Essa solução usa o conceito de "valor anterior" para comparar como o sinal do botão está agora e como ele estava na rodada anterior, para detectar o momento exato em que o pulso foi pressionado (transição de 0 para 1 do sinal). Veja o algoritmo:

```
leia o valor do botao_atual
se o botão_atual == 1 e botao_anterior == 0 ...
        inverta o LED
faça botao_anterior = botao_atual
```

Exercício:

1. Escreva o código para esse algoritmo, e verifique o seu funcionamento no simulador. (OBS: a variável bt_anterior tem que ser estática!)
2. Observe atentamente, na simulação, que a ação de inverter o LED acontece na transição de subida (do 0 para 1) do sinal do botão. Modifique o programa, para que a ação de inverter o LED aconteça somente na transição de descida (de 1 para 0).

Solução 4: usando Máquinas de Estados Finitos, para avaliar a evolução do SISTEMA

Existem várias formas de representar um algoritmo. Nos exemplos anteriores, utilizamos uma forma textual para representar uma sequência de passos, mas às vezes, uma representação gráfica, em forma de um diagrama (por exemplo, um Fluxograma), permite uma melhor visualização do problema. Uma outra forma, que usaremos muito nessa disciplina, é o Diagrama de Estados, que serve para representar uma Máquina de Estados Finitos.

Essa solução usa o conceito de Máquina de Estados Finitos, para generalizar o conceito abordado na solução anterior. A ideia continua a mesma: considerar o "estado anterior" e as "condições" (sinais de entrada), para determinar as ações e o estado seguinte (futuro). A diferença é que, no exemplo anterior, nosso foco estava na evolução do **sinal** (do botão) e, agora, nosso foco está na evolução do **sistema** (do botão).

O Diagrama de Estados, abaixo, foi construído da seguinte forma: você começa sempre desenhando um estado inicial (0). No nosso caso, o estado 0 representa o nosso sistema esperando que o botão seja pressionado, logo, no estado 0, há apenas duas alternativas: se o botão for pressionado (bt==1) então ele executa a ação de inverter o LED, e segue para um próximo estado (1); ou, se o botão não for pressionado (bt==0), ele permanece no estado 0. Depois que você termina de desenhar todas as alternativas desse estado, parte-se para o próximo. No estado 1, o sistema estará esperando que o botão seja liberado e, novamente, há duas alternativas: se o botão for liberado (bt==0), o sistema retorna ao estado 0 (e fica pronto para aguardar o próximo pulso), caso contrário (bt==1), o sistema permanece no estado 1.

Diagrama de Estados

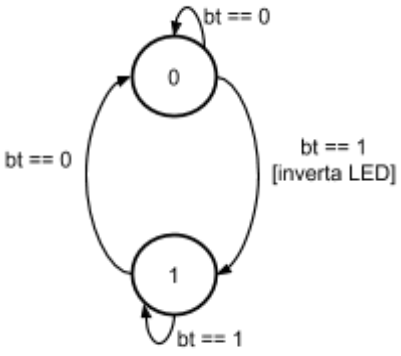


Tabela de transições

Estado Anterior	Condição	Ação	Estado Fututo
0	bt == 0	----	0
0	bt == 1	inverta LED	1
1	bt == 1	----	1
1	bt == 0	----	0

Para esse exemplo, pode parecer que essa solução é "maior", ou mais "burocrática"... mas o fato é que essa solução é muito mais "formal", mais "elegante", e permite modelar problemas muito mais complexos com grande facilidade (isso ficará mais claro em outros exemplos...).

Algoritmo:

```
pelo valor do estado, selecione:
    caso estado seja 0:
        se botão pressionado
            inverta o LED
            faça estado = 1
    caso estado seja 1:
        se botão não pressionado
            faça estado = 0
```

Exercício:

1. Leia sobre a estrutura de controle switch / case e implemente o código para a MEF descrita acima.
2. Escolha uma das soluções acima (exceto a solução 1, que não funciona), e faça com que os 4 botões controlem 4 LEDs, individualmente (botão1 controla o LED1, botão2 controla LED2…)

Vetores e Laços de Repetição (for)

Conceitos:

- vetor (também chamado de array)
- Laço de repetição (for)

No exercício 2 do capítulo anterior, vimos que o código ficou muito repetitivo, tanto na declaração das constantes que representam os pinos onde os LEDs e botões estão conectados, quanto na execução das instruções que descrevem o procedimento para detectar o pulso em cada botão, para decidir o acionamento de cada LED. Esse exemplo é perfeito para demonstrarmos a utilidade dos vetores e laços de repetição:

Exercício:

```
// ao invés de declarar cada LED, individualmente...
// const byte LED1 = 13;
// const byte LED2 = 12;
// const byte LED3 = 13;
// const byte LED4 = 10;

// podemos declarar um vetor, com os pinos de 4 LEDs
const byte LED[] = {13, 12, 11, 10};
```

1. Considere o exemplo acima, e faça o mesmo para a declaração dos botões

```
void setup() {
  // ao invés de repetir a mesma instrução, para cada pino, individualmente,
  // pinMode ( LED1, OUTPUT );
  // pinMode ( LED2, OUTPUT );
  // pinMode ( LED3, OUTPUT );
  // pinMode ( LED4, OUTPUT );

  // podemos criar um laço de repetição:
  for (byte i = 0; i < 4; i++) { // para i variando de 0 a 3,
    pinMode ( LED[i], OUTPUT );  //   configure o LED[i] como SAÍDA
  }
}
```

2. adicione ao código acima a configuração dos pinos dos botões, como entradas.
3. agora, repita o exercício 2, do capítulo anterior, usando o laço (for) para controlar os 4 LEDs, com os 4 botões, sem ter que copiar e colar o mesmo código, 4 vezes.
4. Volte ao exercício 4 da seção Entradas e Saídas Digitais. Use vetores e laços de repetição para fazer com que os LEDs de 0 a 7 sejam controlados por caracteres recebidos pela Serial: o LED[0] acende ao receber o caractere ‘A’, e apaga quando receber o caractere ‘a’, o LED[1] acende ao receber o caractere ‘B’, e apaga quando receber o caractere ‘b’, e assim por diante. DICA: caracteres são números… logo, é possível calcular, por exemplo, que ‘a’ + 1 = ‘b’.
5. Implemente agora um comportamento diferente: o sistema deverá CONTAR quantos botões estão pressionados ao mesmo tempo, e acender somente 1 LED, correspondente a essa contagem. Ex: se 3 botões estiverem pressionados (não importa quais botões), somente o LED 3 deverá estar aceso. Se nenhum botão estiver pressionado, nenhum LED estará aceso. DICA: use laço for para fazer a contagem de quantos botões estão pressionados, e depois, outro laço for para ativar / desativar os LEDs, de acordo com a contagem.

6. No exercício 3 da seção Entradas Analógicas, fizemos com que o giro do potenciômetro, de 0 a 100%, fosse dividido em 3 intervalos, cada intervalo correspondendo a um, dentre 3 LEDs. Use o conceito de vetores para fazer o mesmo, para todos os 8 LEDs. DICA: siga o algoritmo abaixo:

```
faça: pot = valor analógico do potenciômetro
para i variando de 0 a 7, repita:
        se i igual a pot / 128
                acenda LED[i]
        caso contrário
                apague LED[i]
```

7. Na questão anterior, o acionamento do LED indicador foi controlado pelo valor analógico de um potenciômetro. Faça a mesma funcionalidade, agora controlando com dois botões digitais: a cada pulso no botão[1], o LED ativo desloca-se para a direita (+1), e a cada pulso no botão[0], o LED ativo desloca-se para a esquerda (-1).

Funções

Conceitos:

- Leia este tutorial, para compreender os conceitos de:
  - declaração e chamada de função,
  - passagem de parâmetros (argumentos)
  - valor de retorno

Declaração e Chamada:

Considere os programas abaixo:

```
const byte LED1 = 13;

void setup() {
  pinMode ( LED1, OUTPUT );
}

void loop() {
  digitalWrite ( LED1, HIGH );
  delay ( 500 );
  digitalWrite ( LED1, LOW );
  delay ( 500 );
}
```

```
const byte LED1 = 13;

void setup() {
  pinMode ( LED1, OUTPUT );
}

void loop() {
  piscaLED(); // executa (chama) a função piscaLED
}


// declara função piscaLED
void piscaLED() {
  digitalWrite ( LED1, HIGH );
  delay ( 500 );
  digitalWrite ( LED1, LOW );
  delay ( 500 );
}
```

ambos fazem exatamente a mesma coisa: piscar o LED conectado ao pino 13, com período = 1s, entretanto, no programa da esquerda, todas as instruções para fazer o LED piscar foram colocadas diretamente dentro da função loop()... já no programa da direita, foi criada uma função com esse detalhamento, com o sugestivo nome 'piscaLED', de forma que, na função loop(), ficou somente a chamada para a função piscaLED(). Claro que nesse exemplo, extremamente simples, a vantagem de fazer isso é questionável… mas imagine um programa maior, mais complexo: ele se tornará muito mais legível se nós o dividirmos em funções, com nomes apropriados, como 'piscaLED', 'invertaLED', 'pulsoNoBotao'... de modo a esconder os detalhes dentro dessas funções, e deixar a função principal mais legível.

Exercício:

1. Em exemplos anteriores, já vimos que podemos inverter o estado de um LED com o uso da sentença: `digitalWrite (LED1, ! digitalRead(LED1));`
   a. Declare uma função, com o nome inverteLED(), que inverta o estado do LED1 a cada chamada.
   b. Modifique a função piscaLED(), dada acima, para que ela use a inverteLED().

Passagem de Parâmetros (Argumentos)

A função piscaLED(), acima, não recebe nenhum parâmetro, e age sempre sobre o LED1, que pisca com tempo fixo de 500ms, porque esses valores foram inseridos diretamente em seu código. Podemos tornar essa função mais flexível, podendo agir sobre qualquer LED, fazendo-o piscar com qualquer tempo, através do uso de parâmetros. Compare:

```
void loop() {
  piscaLED(); // sem parâmetros
}


// declara função piscaLED
// nenhum parâmetro
void piscaLED() {
  digitalWrite ( LED1, HIGH );
  delay ( 500 );
  digitalWrite ( LED1, LOW );
  delay ( 500 );
}
```

```
void loop() {
  piscaLED(12, 300); // passando parâmetros
}
// declara função piscaLED
// recebe como parâmetros:
// byte pinoLED: qual o pino do LED que deve piscar
// int tempo: qual o tempo de cada estado
void piscaLED(byte pinoLED, int tempo) {
  digitalWrite ( pinoLED, HIGH );
  delay ( tempo );
  digitalWrite ( pinoLED, LOW );
  delay ( tempo );
}
```

Enquanto a função **`piscaLED()`**, da esquerda, está limitada a piscar sempre o LED1 (pino 13), com tempo = 500, a função **`piscaLED(byte pinoLED, int tempo)`**, da direita, recebe dois parâmetros (argumentos): o primeiro será o pino que deverá piscar (pinoLED), e o segundo será o tempo de cada estado aceso/apagado, do LED (tempo). Desta forma, a chamada **`piscaLED(12, 300);`** dentro da função loop(), fará piscar o LED no pino 12, com tempo = 300ms.

Exercícios:

2. Use a função **`piscaLED(byte pinoLED, int tempo)`**, dada acima, para fazer com que o LED 13 pisque com tempo variável, controlado pelo valor analógico do potenciômetro.
3. Ainda usando a função **`piscaLED(byte pinoLED, int tempo)`**, retorne ao exercício 6 do capítulo Vetores e Laços de Repetição (for) , e faça com que o LED correspondente ao valor do potenciômetro pisque com período fixo, de 1s.
4. Ainda usando a função **`piscaLED(byte pinoLED, int tempo)`**, retorne ao exercício 7 do capítulo Vetores e Laços de Repetição (for) , e faça com que os botões [0] e [1] selecionem qual LED irá piscar, e os botões [2] e [3] ajustem o tempo.
5. Declare uma função **`inverteLED (byte pinoLED)`**, onde o parâmetro pinoLED indica qual o pino cujo estado será invertido.

Valor de Retorno:

Considere novamente o problema de detectar Um simples pulso num botão… naquele capítulo, vimos várias soluções para inverter o estado de um LED a cada transição de subida (de 0 para 1) no sinal gerado por um botão. Ok, mas… e se eu não quiser mais que a ação gerada a partir do pulso no botão seja a de "inverter um LED"? e se a ação desejada for incrementar uma contagem? ou decrementar? ou registrar o dígito de uma senha? ou lançar um míssil balístico??? - ou seja: seria interessante ter uma função que detecte se houve ou não um pulso no botão, mas que essa função não realize ação nenhuma - apenas **retorne** um valor para que o programa principal realize a ação que desejar. Assim, teremos uma função genérica, que servirá tanto para inverter um LED, quanto para realizar uma contagem, ou para lançar um míssil… Vamos ver como seria isso:

```
// Declarações Globais
const byte LED1   = 13;          // declara constante LED1   = 13
const byte BOTAO1 =  5;          // declara constante BOTAO1 =  5

void setup() {
  pinMode ( LED1, OUTPUT );      // configura pino 13 como SAÍDA
  pinMode ( BOTAO1, INPUT );     // configura pino  5 como ENTRADA
  Serial.begin( 9600 );          // inicializa comunicação serial a 9600 bits/s
}

void loop() {
  delay (10);                    // (somente para simulação)
  if ( pulsoNoBotao() == 1 ) {       // se a chamada à função pulsoNoBotao retornar 1...
                                 //   realize a ação desejada!
                                 //   (nesse caso, inverte o LED1, mas poderia ser
                                 //   qualquer outra coisa...)
    digitalWrite (LED1, ! digitalRead(LED1));
  }
}

// declaração da função pulsoNoBotao
// Observe agora o uso do tipo bool: isso indica que essa função RETORNA um valor
// booleano ao final de sua execução
bool pulsoNoBotao() {
  bool pulso = 0;                // memoriza se houve pulso (1) ou não (0)
  static bool bt_anterior = 0;   // memoriza estado anterior do botão
  bool bt = digitalRead(BOTAO1); // lê estado atual do botão
                                 // se houve transição de 0 para 1 no botão,
  if ( bt == 1 && bt_anterior == 0 ) {
                                 //   aqui executava a ação de inverter o LED, lembra?
    // digitalWrite (LED1, ! digitalRead(LED1);
    pulso = 1;                   // em vez disso, apenas marca 1 no valor de retorno
  }
  bt_anterior =  bt;
  return pulso;                  // retorna valor de pulso (0 ou 1)
}
```

Exercícios:

6. Reescreva a função pulsoNoBotao(), mas agora com base na Solução 4: usando Máquinas de Estados Finitos, de forma que ela também retorne o valor 1 somente quando ocorrer a transição de 0 para 1 no sinal do botão.
7. Usando agora o conceito de Vetores, declare novas funções, conforme pedido abaixo:
    a. **`bool inverteLED(byte n)`**: essa função deverá inverter o estado do LED[n], e retornar o estado atual do LED[n] (após a inversão).
    b. **`bool pulsoNoBotao(byte n)`**: essa função deverá verificar o BOTAO[n] e retornar 1 se houve transição 0 para 1 neste botão, ou 0, caso contrário.

Temporização não bloqueante

Até aqui, temos usado a função **`delay()`** para temporização de nossos programas… o problema é que essa função é bloqueante, ou seja, ela faz com que o programa fique travado durante todo o tempo até que o delay acabe, e isso é indesejado quando seu sistema é multitarefa, porque todas as tarefas ficarão travadas quando uma delas executar um **`delay()`**. Para contornar esse problema, o Arduino dispõe da função **`millis()`**, que não bloqueia a execução do programa. Funciona assim: o Arduino tem um "relógio" interno, que conta quantos milissegundos se passaram desde que o mesmo foi ligado. Esse relógio está contando, independente a execução do seu programa… quando a função **`millis()`** é chamada, ela simplesmente retorna o valor desse contador. Caberá a você, programador, usar esse valor para decidir se está no "tempo" de executar a ação desejada. Vejamos um exemplo:

Considere o botão liga/desliga do seu smartphone, ou do PC: com um pulso curto, ele liga o aparelho, e com um pulso longo (mantendo-o pressionado por cerca de 2s, ou mais), ele desliga. Como implementar essa funcionalidade no Arduino, de forma que ele acenda ou apague um LED com um pulso curto ou longo de um botão?

Vou demonstrar aqui uma solução, usando MEF. Claro que essa não é a única solução, mas acredito ser a forma mais fácil de abordar o problema. Se você quiser tentar outras formas, desejo-lhe boa sorte, e que me traga o resultado!

Vamos à descrição, passo a passo, do raciocínio que nos leva à construção do Diagrama de Estados:

A cada rodada do programa, atualizaremos as variáveis **`ta = millis();`** e **`b = digitalRead(BOTAO);`**

Enquanto o botão não for pressionado (b==0), o sistema permanecerá em seu estado inicial (0). Quando o botão for pressionado, o sistema fará a inicialização da contagem do tempo (t0 = ta), e então passará para o estado seguinte (1).

No estado 1, faremos a contagem do tempo que o botão permanece pressionado. Aqui, há duas condições de saída: se o botão permanecer pressionado por muito tempo, a diferença ta - t0 vai crescendo continuamente até que, chegando ao limite estabelecido (ta - t0 >= 2000), considera-se que houve um pulso "longo": o LED será apagado, e o sistema vai para o estado 2. Caso contrário, se o botão for liberado antes da contagem atingir o limite ta - t0 >= 2000, considera-se que foi dado um pulso "curto": o LED será aceso, e o sistema volta ao estado inicial (0), para aguardar um novo pulso.

O estado 2 é necessário para que o sistema espere a liberação do botão (b==0) após o pulso longo, antes de voltar ao estado inicial (0).

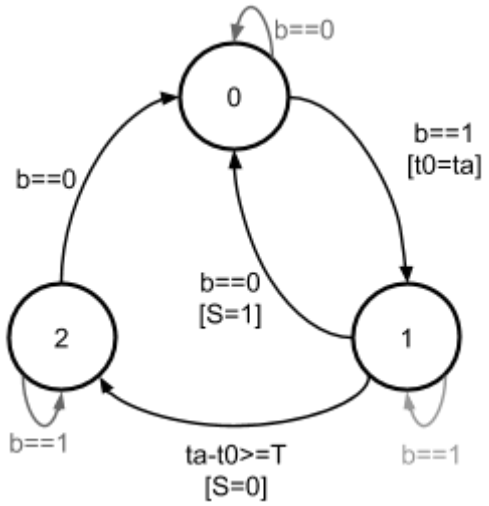


Diagrama de Estados

Estado Atual	Condição	Ação	Estado Futuro
0	b==0	---	0
0	b==1	t0 = ta	1
1	b==0	S=1	0
1	b==1	---	1
1	ta-t0>=T	S=0	2
2	b==0	---	0
2	b==1	---	2

Tabela de Transições

A seguir, vemos o Algoritmo (à esquerda) e o código (à direita), correspondentes à MEF descrita acima. Note que o código não está completo: contém estritamente a funcionalidade correspondente ao algoritmo, mas você precisa complementar com as declarações iniciais, funções setup() e loop()...

pelo valor de (estado), selecione:
caso 0:
se botão pressionado,
faça t0 = ta;
faça estado = 1;

caso 1:
se botão não pressionado,
ative a Saída;
faça estado = 0;

caso contrário, se ta-t0>=T,
desative a Saída;
faça estado = 2;

caso 2:
se botão não pressionado,
faça estado = 0;

```
static byte estado = 0;
static unsigned long t0 = 0;
unsigned long ta = millis();
bool b = digitalRead(BOTAO);

switch (estado) {
  case 0:
    if (b == 1) {
      t0 = ta;
      estado = 1;
    }
  break;

  case 1:
    if (b == 0) {
      digitalWrite (S, HIGH);
      estado = 0;
    }

    else if (ta - t0 >= T) {
      digitalWrite (S, LOW);
      estado = 2;
    }
  break;

  case 2:
    if (b == 0) {
      estado = 0;
    }
  break;
}
```

Exercícios:

1. Implemente o código completo (declarações globais, funções setup e loop...) para que essa funcionalidade possa ser testada na prática.

2. Mova toda a funcionalidade da MEF para uma função, **`void clickCurtoLongo(byte i)`**, onde i (de 0 a 3) seja o índice que permita selecionar um conjunto botão/LED, em vetores com 4 botões e 4 LEDs. Implemente a função loop(), de modo que os 4 botões controlem os 4 LEDs, “simultaneamente”.

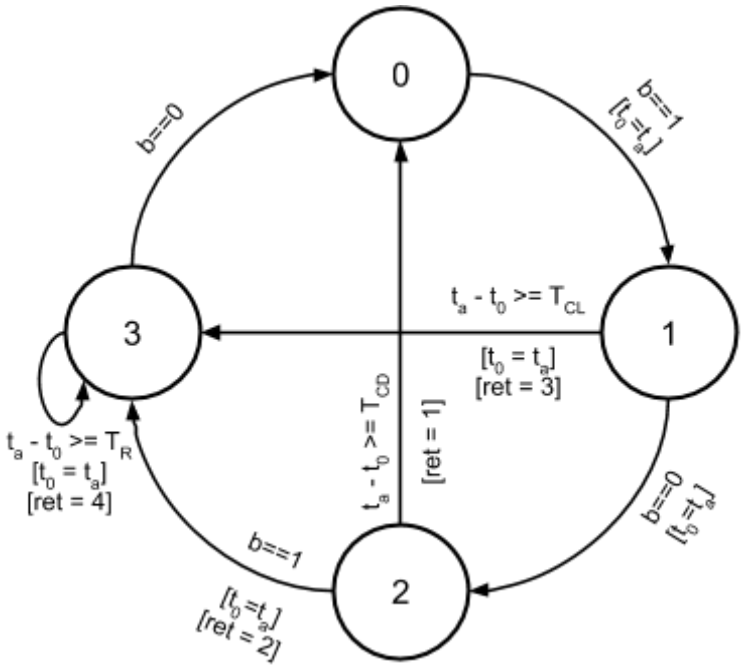
3. A função acima mistura duas funcionalidades: a de identificar se o pulso é curto ou longo, e a de acionar o LED, de acordo com a duração do pulso. Já vimos que isso não é uma boa prática. Modifique a função anterior para que, em vez de acionar o LED, ela apenas retorne um código, para indicar o comportamento detectado no botão. Caberá ao programa principal, com esse código retornado, decidir qual ação tomar. A função deverá retornar os códigos a seguir:
   0 - nenhum evento detectado no botão.
   1 - pulso curto detectado
   2 - pulso longo detectado

4. Agora, considere a funcionalidade que temos no uso do botão do mouse (ou nos dispositivos sensíveis ao toque), onde um toque “simples”, faz uma coisa, um toque “duplo”, faz outra. Siga todo o procedimento descrito acima, para desenvolver uma nova MEF para essa funcionalidade. Implemente-a em uma nova função: **`byte clickSimplesDuplo(byte i)`**, que deverá funcionar de forma semelhante à desenvolvida na questão 3, mas agora os códigos serão interpretados assim:
   0 - nenhum evento detectado no botão.
   1 - pulso simples detectado
   2 - pulso duplo detectado

5. Agora, considere o comportamento do teclado do PC (suponha que você esteja usando um editor de textos): ao pressionar uma tecla qualquer, o caractere correspondente vai aparecer na tela. Se a tecla continuar pressionada por algum tempo (1s, digamos), esta entra em modo de repetição, e o caractere será repetido na tela, com uma certa velocidade (a cada 200ms, digamos). Siga todo o procedimento descrito acima, para desenvolver uma nova MEF para essa funcionalidade. Implemente-a em uma nova função: **`byte clickRepete(byte i)`**, que deverá funcionar de forma semelhante à desenvolvida na questão 3, mas agora os códigos serão interpretados assim:
   0 - nenhum evento detectado no botão.
   1 - o botão foi pressionado nesse momento
   2 - o botão entrou em modo de repetição, e esse código será retornado a cada 200ms (intercalado com 0...)

6. Combinando as funcionalidades das questões 4 e 5, implemente uma nova função: **`byte eventoBotao(byte i)`**, que detecta o tipo de evento ocorrido em um botão, retornando os códigos abaixo:
   0 = nenhum evento detectado
   1 = click simples (1 pulso com duração menor que TEMPO_CLICK_LONGO)
   2 = click duplo (2 pulsos em tempo menor que TEMPO_CLICK_DUPLO)
   3 = click longo (1 pulso com duração maior que TEMPO_CLICK_LONGO)

   4 = modo repetição (após click longo, persistindo botão pressionado, emite esse código a cada intervalo TEMPO_REPETE)




```
// função eventoBotao()
// detecta o tipo de evento ocorrido em um botão
// retorna:
//   0 = nenhum evento detectado
//   1 = click simples
//   2 = click duplo (2 clicks em tempo menor que TEMPO_CLICK_DUPLO)
//   3 = click longo (1 click com duração maior que TEMPO_CLICK_LONGO)
//   4 = modo repetição (após click longo, persistindo botão pressionado,
//                       emite esse código a cada intervalo TEMPO_REPETE)
byte eventoBotao() {
  static byte estado = 0;
  static unsigned long t0 = 0;
  unsigned long ta = millis();

  byte ret = 0;
  bool bt = digitalRead(BOTAO1);

  switch(estado) {

    case 0:
    if (bt) {
      t0 = ta;
      estado = 1;
    }
    break;

    case 1:
    if (!bt) {
      t0 = ta;
      estado = 2;
    }
    else if (ta - t0 >= TEMPO_CLICK_LONGO) {
      t0 = ta;
      ret = 3;   // click longo
```

```
      estado = 3;
    }
    break;

    case 2:
    if (bt) {
      t0 = ta;
      ret = 2;    // click duplo
      estado = 3;
    }
    else if (ta - t0 >= TEMPO_CLICK_DUPLO) {
      ret = 1;    // click simples
      estado = 0;
    }
    break;

    case 3:
    if (!bt) {
      estado = 0;
    }
    else if (ta - t0 >= TEMPO_REPETE) {
      t0 = ta;
      ret = 4;     // modo repetição
    }
    break;
  }
  return ret;
}
```