Apostila de Microcontroladores (Hardware)

Tiago F. Tavares

2017

Sumário

Capítulo 1

Algoritmos

1.1 Máquinas de Estado

Uma máquina de estado é uma abstração, isto é, um modelo que pode ser usado para analisar determinados tipos de sistemas. Os sistemas aos quais esse modelo se aplica são aqueles que, a cada instante de tempo, assumem um, e apenas um, dentre N estados possíveis. Para cada estado, o sistema assume um comportamento diferente e, dependendo de suas entradas, transiciona para outro estado.

Poderíamos analisar o vôo de um avião como uma máquina de estados. Inicialmente, ocorre o estado de embarque. Depois, há o estado de decolagem. Após, o estado de vôo de cruzeiro, o pouso e, por fim, o desembarque. Perceba que o comportamento do avião é diferente em cada um desses estados. A transição entre os estados depende da aquisição de dados dos sensores de bordo (GPS, altímetro, velocímetro, etc.) e da comparação desses dados com um plano de vôo.

Para implementar uma máquina de estados em software, precisamos de uma variável para representar (armazenar) o estado em que nos encontramos. Também, precisamos de uma função que implemente as regras de transição de estados. O exemplo mostrado na Figura 1.1 implementa uma máquina de dois estados (0 e 1) que muda de estado em todos os acionamentos.

```
int estado = 0; /* Estado atual da maquina */

void maq_estados() {
  /* Esta implementacao usa uma estrutura de if-then-else explicita */
  if (estado == 0) {
    estado = 1;
  }
  else {
    estado = 0;
  }
}
```

Figura 1.1: Exemplo de implementação de máquina de estado que muda de estdo a cada chamada de `maq_estados()`.

1.2 Scheduler

O problema com o qual lidamos neste estudo de caso é o de controlar múltiplas tarefas em tempo real. Suponha um sistema de monitoramento uma lista de tarefas a fazer:

1. A cada 2s, verificar o estado (aberto/fechado) de cada uma das 5 portas de uma construção;
2. Garantir que as luzes do corredor principal acendam por 15s após um botão ser pressionado;
3. A cada 30s, mandar um sinal de "estou vivo" que indica à central que o dispositivo está funcionando;

Neste problema, estou supondo que tenho sensores que indicam o estado das portas através de tensões em níveis lógicos adequados, acessados por uma função específica (que já foi definida anteriormente) e também que o botão que controla as luzes do corredor é "ideal", no sentido de prover nível lógico 1 quando pressionado e 0 quando solto, sem ruídos. Também, estou supondo que o procedimento para mandar o sinal de "estou vivo" envolve somente uma chamada de função.

Executar somente a tarefa 1 é bastante simples. Posso programar meu microcontrolador para gerar uma interrupção a cada 2 segundos e então verificar o estado de cada uma das portas usando entradas digitais. Em termos de temporização, é bastante semelhante a fazer um LED piscar.

Se queremos executar a tarefa 1 e a tarefa 3 simultaneamente, temos um problema um pouco mais difícil. Poderíamos, em teoria, programar duas interrupções diferentes (uma para a verificação das portas e a outra para enviar o sinal de "estou vivo"). Esta idéia usa um dispositivo de hardware (um timer) para cada uma das tarefas, e, portanto, esse raciocínio limita nossa possibilidade de executar tarefas ao número de timers físicos existentes no sistema. Portanto, pode ser uma boa idéia usar o mesmo timer para controlar mais de uma tarefa.

Isso pode ser feito através de um sistema simples de *scheduling*, ou seja, de agendamento de tarefas. A idéia por trás do scheduling é que temos variáveis que contam unidades de tempo real e vinculamos nossos eventos a estados dessas variáveis. Em nossa rotina de interrupção periódica, poderíamos ter um contador que indica quantas vezes o intervalo de tempo da interrupção se passou - em nosso caso, 2s. Portanto, sempre que esse contador for um múltiplo de 15, sabemos que se passaram 30s, e, portanto, é tempo de enviar a mensagem de "estou vivo":

Note bem que o código para verificar o estado das portas só não depende de um contador porque ele executa exatamente no mesmo período da interrupção.

Uma outra forma de fazer a mesma tarefa seria reiniciar o contador a cada ciclo:

A vantagem da idéia do contador reiniciável é que ela pode ser facilmente adaptada para incorporar, também, a tarefa 2 em nossa rotina. Nesse caso, precisaremos de uma rotina de interrupção com período de 1s, já que este é o maior divisor comum de 2, 15 e 30. A tarefa 1 também teria que ser vinculada a um contador.

```
void ISR_interrupcao_a_cada_2s() {
  static int contador=0;
  verifica_estado_das_portas();
  contador++;
  if ( (contador%15) == 0 ) {
    envia_estou_vivo();
  } else contador++;
}
```

Figura 1.2: Scheduler com contador non-stop.

```
void ISR_interrupcao_a_cada_2s() {
  static int contador=0;
  verifica_estado_das_portas();
  if (contador >= 15) { /* Posso mudar esse valor dinamicamente! */
    envia_estou_vivo();
    contador = 0;
  } else contador++;
}
```

Figura 1.3: Scheduler com contador reiniciado.

Desta forma, teríamos 3 contadores. Dois deles, relacionados às tarefas 1 e 3, seriam reiniciados sempre que atingissem um determinado valor. O outro, relacionado à tarefa 2, seria reiniciado sempre que um botão fosse pressionado.

A detecção do pressionar do botão do corredor, porém, também deve ser controlada por nossa rotina. Parece razoável que o estado do botão seja verificado a cada 10ms - dificilmente alguém apertaria um botão mais rápido que isso. Então, a rotina de interrupção deverá ser executada a cada 10ms e o valor máximo de nossos contadores têm que ser re-calculados:

$$\begin{aligned} 2s &= 200 \times 10ms \\ 15s &= 1500 \times 10ms \\ 30s &= 3000 \times 10ms \end{aligned} \tag{1.1}$$

Portanto, nosso código ficaria como mostra a Figura 1.4.

Desta forma, mostro que é possível (e desejável) agregar diversas tarefas em uma única rotina de interrupção. Além disso, essa estrutura permite mudar, dinamicamente, os períodos de realização de cada rotina, simplesmente atribuindo novos valores às variáveis max_contadorN.

Um aspecto importante da tarefa 2 é que, nesse caso, existem dois eventos a monitorar: a passagem do tempo de acendimento da lâmpada e o pressionar do botão. Veja como esses eventos interagem utilizando a variável contador2.

```
void ISR_interrupcao_a_cada_10ms() {
  /* Cronometros */
  static int contador1=0; /* Tarefa 1 */
  static int contador2=1500; /* Tarefa 2 */
  static int contador3=0; /* Tarefa 3 */

  /* Limites maximos */
  static int max_contador1 = 200;
  static int max_contador2 = 1500;
  static int max_contador3 = 3000;

  /* Tarefa 1 */
  if (contador1 >= max_contador1) {
     verifica_estado_das_portas();
     contador1 = 0;
  } else {
     contador1++;
  }

  /* Tarefa 3 */
  if (contador3 >= max_contador3) {
    envia_estou_vivo();
    contador3 = 0;
  } else {
    contador3++;
  }

  /* Tarefa 2 */
  if (botao_apertado()) { /* Verifico o botao em todos os ciclos */
     contador2 = 0; }  /* Reiniciar este processo depende do estado do botao
         */

  if (contador2 >= max_contador2) {
     luz_desligada();
  } else {
     luz_ligada();
     contador2++;
  }
}
```

Figura 1.4: Scheduler controlando três tarefas distintas.

Capítulo 2

Componentes

2.1 E/S de propósito geral

As portas de entrada e saída de propósito geral (GPIO) de um microcontrolador permitem converter dados (1 e 0) em tensões físicas e vice-versa. Os pinos (físicos) de uma porta GPIO se relacionam a bits (lógicos) que podem ser acessados em programas que executam no microcontrolador. As portas GPIO podem ser configuradas como *entrada* com com *saída*, como descreve a Tabela 2.1.

Modo	Operação	Características
Entrada	Converte uma tensão alta aplicada no pino em um bit 1 na saída, e vice-versa, numa posição de memória conhecida.	Alta impedância de entrada. Pode receber tensões dentro de limites conhecidos.
Saída	Converte um bit 1 em uma tensão alta no pino e vice-versa.	Baixa impedância de saída. Pode ser modelado como uma fonte de tensão. Pode fornecer corrente dentro de limites conhecidos.

Tabela 2.1: Modos de operação de um GPIO.

Muitos microcontroladores modernos têm um registrador que controla a direção dos pinos, e outro registrador – que deve ser lido ou escrito, de acordo com a direcionalidade do pino – que se relaciona ao estados (tensão alta ou baixa) dos pinos. As correspondências entre os conectores físicos do microcontrolador e suas representações lógicas podem ser encontradas no manual do usuário do microcontrolador. Os limites elétricos (tensão e corrente máximas e mínimas) das portas de entrada e saída variam significativamente de dispositivo para dispositivo e devem ser consultados antes de começar o processo de planejamento e montagem.

2.2 Conversor A/D

O conversor analógico-digital (ou simplesmente A/D) é um dispositivo que recebe, na entrada, uma tensão elétrica e produz, na saída, uma representação digital para essa mesma tensão, como mostra a Figura 2.1. É comum que essa representação digital seja um inteiro sem sinal, com um número b de bits que varia de acordo com o dispositivo e as suas configurações. Assim, um conversor A/D é capaz de representar 2^b valores de tensão diferentes.

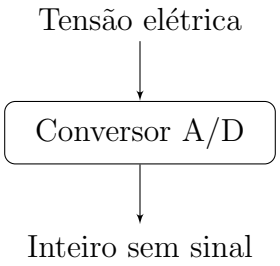

Figura 2.1: Processo de conversão analógico-digital.

Os valores de tensão aceitos pelo conversor A/D devem estar compreendidos entre dois valores de referência (V_- e V_+). As tensões entre os valores de referência são divididos em 2^b faixas, sendo cada uma associada a um possível número na saída num processo chamado de quantização. Entradas fora dos limites de referência levam à saturação do processo de conversão. Isso significa que o aumento do número de bits do conversor A/D leva à redução do erro de quantização, já que diminui a faixa de valores de tensão associados a cada número.

O conversor A/D demora um certo tempo para realizar o processo de conversão. Isso acontece porque o processo de conversão A/D envolve executar um algoritmo de aproximação, o que toma um intervalo de tempo que varia de acordo com o dispositivo de conversão e suas configurações. Isso significa que o processo de conversão só pode ser realizado um certo número f_s de vezes por segundo. Esse número f_s é chamado de *frequência de amostragem*.

2.3 Modulação de Largura de Pulso

A modulação por largura de pulso (Pulse Width Modulation – PWM) é uma técnica de engenharia que permite controlar intensidades de manifestações analógicas usando apenas níveis discretos (lógicos) de sinal. Isso pode ser entendido como uma forma de conversão digital-analógico. PWMs permitem controlar, por exemplo, a velocidade de motores DC ou a luminosidade de LEDs ou lâmpadas.

A teoria subjacente ao funcionamento do PWM é bem conhecida e amplamente divulgada em comunidades ligadas à eletrônica. PWMs partem da idéia de gerar um sinal periódico que assume apenas valores discretos (0 e Vcc). Esse sinal deve ter uma frequência muito superior à frequência de operação do dispositivo que se deseja controlar. A intensidade da operação do dispositivo controlado é proporcional ao duty-cycle (razão entre o tempo de alto e o período do sinal gerado). A Figura 2.2 mostra sinais de mesma frequência, mas com diferentes duty-cycles.

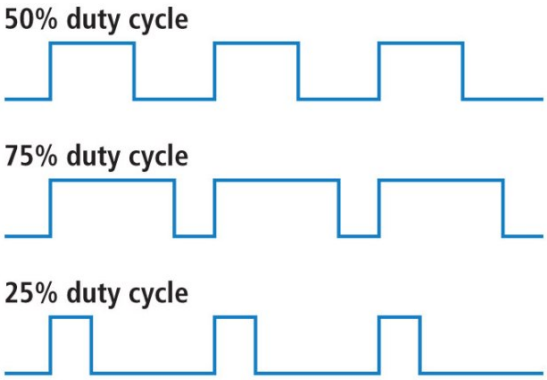

Figura 2.2: Sinais de mesma frequência e diferentes duty-cycles. Figura extraída de [**?**].

Essa forma de controle é eficiente para modular a luminosidade de LEDs e lâmpadas, a velocidade de motores DC e outros dispositivos com comportamento tipicamente inercial. Trata-se de uma operação semelhante a ligar e desligar um interruptor muitas vezes por segundo, bem mais rápido que se possa observar o dispositivo voltando ao estado de repouso. Quanto maior a proporção de tempo em que o interruptor está ligado, maior será a luminosidade observada.

Podemos mostrar que a potência entregue a uma carga resistiva por um sinal PWM de período T é proporcional a seu duty-cycle w. Lembramos que a potência entregue por uma fonte de tensão DC (com tensão constante v) a uma carga resistiva com resistência R é dada por:

$$P_0 = \frac{1}{T}\int_0^T \frac{v^2}{R}dt = \frac{v^2}{RT}\int_0^T dt = \frac{v^2}{R}. \tag{2.1}$$

Se temos, no lugar de uma tensão DC, um sinal PWM com duty-cycle igual a w, então sabemos que a tensão é igual a zero em todos os instantes entre wT e T, de forma que a expressão 2.1 pode ser modificada para:

$$P = \frac{1}{T}\int_0^{wT} \frac{v^2}{R}dt = \frac{v^2}{RT}\int_0^{wT} dt = \frac{v^2}{R}w = wP_0. \tag{2.2}$$

Veja que isso permite controlar linearmente a potência entregue à carga. Se, de outra maneira, tivéssemos optado por multiplicar a magnitude da tensão fornecida à carga por um fator q, teríamos:

$$P = \frac{(qv)^2}{R} = q^2\frac{v^2}{R} = q^2P_0. \tag{2.3}$$

Desta forma, a potência entregue a carga depende quadraticamente de q. Portanto, o PWM permite que a resposta do controle seja linear em relação ao sinal, o que pode facilitar a construção de modelos em diversas situações de controle.

Em microcontroladores modernos, há dispositivos de hardware já preparados para gerar sinais PWM. O Processor Expert já disponibiliza componentes capazes de gerar sinais PWM, que podem ser utilizados livremente nesta montagem.

2.4 Comunicação Digital

A função de um protocolo de comunicação é garantir que uma informação seja transmitida corretamente entre dois dispositivos. Em verdade, já fizemos isso durante este curso. Vamos tomar, como exemplo, o nosso botão de pressão (*push button*).

A informação que ele carrega pode ser representada por um bit. Usando o GPIO, podemos fazer essa informação ser transmitida para a memória do microcontrolador para que possa ser usada num processo de tomada de decisões. Veja que, neste caso, tivemos que projetar antecipadamente:

1. A forma física em que a conexão aconteceria (neste caso, por meio de uma conexão elétrica);

2. Os níveis de tensão correpondentes aos bits 1 e 0.

Da mesma forma, poderíamos configurar uma saída digital (via GPIO) que obedecesse às mesmas configurações de tensão e conectividade. Assim, poderíamos usar um outro dispositivo – desta vez, microcontrolado – que emulasse o botão.

Um outro problema é o de transmitir um byte (e não mais um bit) de um dispositivo para outro. Poderíamos usar um sinal analógico para fazer isso (por meio de conversão D/A e então A/D), mas sinais analógicos estão sujeitos a ruído e atenuação, que são fatores que gostaríamos de evitar. Uma solução possível é utilizar a comunicação assíncrona.

Neste caso, os dois dispositivos envolvidos no processo de comunicação acertam, antecipadamente, a taxa de transmissão de bits (BAUD) em que a comunicação será executada. Uma vez que conhecemos a taxa de transmissão de bits e a quantidade de bits em um byte, é possível projetar um dispositivo que lê, sucessivamente, os bytes enviados por um outro dispositivo.

Veja que, neste caso, devemos definir também um estado de repouso para a linha de transmissão, uma forma de anunciar que o próximo byte será transmitido e uma forma de anunciar que o byte transmitido terminou. O protocolo RS-232 [**?**] define esses elementos todos, que devem ser acertados em todos os dispositivos envolvidos na comunicação antes de iniciar o processo de transmissão de dados. É comum usar as configurações 9600,8N1, isto é, transmissão de 9600 bits por segundo, 8 bits de dados, bit de paridade (usado para correção de ímpar) nenhum e 1 bit de parada.

O protocolo RS-232 define que dados serão transmitidos num conector chamado TX e recebidos num outro, chamado RX. Assim, os conectores TX e RX dos dispositivos envolvidos na comunicação devem se conectar alternadamente, como mostra a Figura 2.3.

Resolvemos, até o momento, o problema de transmitir um byte de um dispositivo a outro. Veja que nossa solução tem duas camadas: uma relacionada ao bit e outra relacionada ao byte. Assim, se encontrásse-mos outra maneira de fazer um bit propagar de um ponto a outro, poderíamos utilizar exatamente os mesmos algoritmos para controlar o fluxo de informação de bytes.

Essa divisão entre as etapas de transmissão deu origem à idéia de *camadas de protocolo* [**?**]. Para transmitir um bit, é necessário definir a camada *física*; a transmissão de um byte é definida

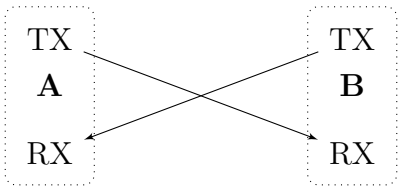

Figura 2.3: Ligações TX-RX no protocolo RS-232, considerando dois diferentes dispositivos A e B.

na camada de *enlace*. Para a comunicação ponto-a-ponto, que utilizaremos nesta montagem, podemos ignorar algumas camadas comumente usadas, ligadas ao endereçamento de pacotes de dados em uma rede. Passaremos diretamente à camada de aplicação, que é responsável por gerar mensagens que significam alguma coisa no contexto de uma aplicação específica. Como mostra a Figura 2.4, nosso protocolo será composto de três camadas.

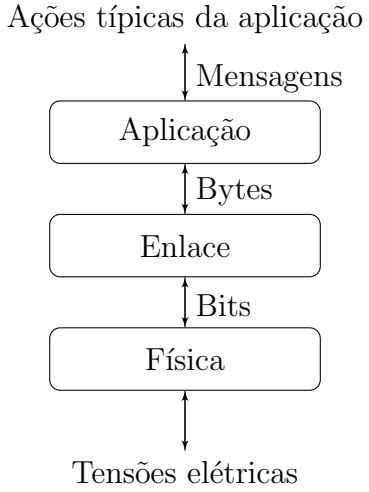

Figura 2.4: Camadas de protocolo na comunicação ponto-a-ponto que será implementada neste roteiro.

Veja que o MCU que utilizamos já implementa a camada de enlace, e, portanto, este problema já está automaticamente resolvido. O periférico que implementa esta camada se chama UART – Universal Asynchronous Receiver-Transmitter.

Também, utilizaremos um dispositivo que converte mensagens RS-232 para Bluetooth. Ele substitui as camadas de enlace e física por equivalentes sem fio. Veja que as informações que transmitimos – isto é, a camada de aplicação – permanecerá a mesma.

2.5 Teclado Matricial

Um teclado matricial é um dispositivo que organiza teclas na forma de uma matriz – com linhas e colunas. Há um contato externo para cada linha e um contato externo para cada coluna. Em estado de repouso, a impedância entre os contatos das linhas e das colunas é muito alto. Quando a tecla na posição (i, j) é pressionada, a impedância entre os contatos da linha i e da coluna j cai para próximo de zero. Se nenhuma ou apenas uma tecla estiver sendo pressionada, a

impedância entre os contatos de duas linhas distintas (assim como de duas colunas distintas) é muito alto.

Usando essa propriedade, é possível executar uma rotina chamada varredura. Essa rotina se baseia numa ligação segundo a qual os contatos das linhas estão ligados a saídas do GPIO de um microcontrolador, e as colunas estão ligadas a entradas. Uma modificação na polaridade do sinal só pode chegar da linha i à coluna j se a impedância entre i e j for baixa, ou seja, se a tecla (i, j) estiver pressionada. A rotina pode ser implementada usando o pseudo-código mostrado na Figura 2.5.

```
função Varredura():
  Para cada pino de saída x[i]:
    Configura x[i] como ativo;
    Para cada pino de entrada y[j]:
      Se y[j] está ativo:
        retorna caractere na posição (i,j);
    Configura x[i] como inativo;
```

Figura 2.5: Pseudo-código de uma rotina de varredura.

A rotina de varredura se baseia, implicitamente, na idéia de que o não-contato em cada entrada relacionada à coluna j implica na leitura de um estado padrão. Para isto, é necessário que um pino de entrada nunca seja deixado flutuando, isto é, sem conexão nenhuma. Esse problema pode ser resolvido usando uma configuração de resistor pull-up, mostrada na Figura 2.6.

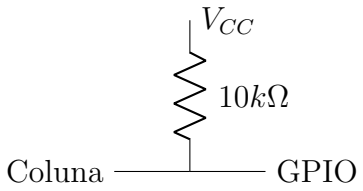

Figura 2.6: Ligação com resistor de pull-up.

Veja que, nesse caso, se a conexão da coluna não existir, então a leitura na GPIO será, forçadamente, um nível alto. Portanto, o nível de tensão padrão – inativo, no pseudo-código da Figura 2.5 – equivale à leitura do nível de tensão alto. O microcontrolador que utilizamos no laboratório permite configurar resistores de pull-up internos, usando um componente Init_GPIO.

2.5.1 De-bouncing

Chaves mecânicas apresentam um problema que se chama *bouncing* [**?**]. Ao ser pressionada, a chave oscila algumas vezes entre as posições aberta e fechada. Portanto, ao ler um caractere válido, é preciso evitar leituras do teclado por algum tempo. A inibição da leitura pode ser realizada através de um algoritmo de scheduling, como o mostrado na Figura 2.7.

```
função interrupção_periodica():
  Se não estou em debouncing:
    A = Varredura();
    Se A é um caractere válido:
      camada_de_aplicação(A);
      mudo para modo debouncing;
      contador_debouncing recebe máximo;
  Se estou em debouncing:
  // Processo análogo ao scheduling
    Decremento contador_debouncing;
    Se contador_debouncing é zero:
      Saio do modo debouncing
```

Figura 2.7: Pseudo-código para a rotina de *debouncing*, incluindo sua relação com a rotina de varredura.

2.6 EEPROM externa com protocolo I2C

EEPROM (Electric Erasable and Programmable Read-Only Memory) é um dispositivo de memória não-volátil. Nesta seção, discutiremos um componente específico – AT24C16 – e seu funcionamento· Trata-se de um circuito integrado de memória cujo acesso é realizado através de protocolo I2C (Inter-Integrated Circuit). Esse protocolo pode ser discutido da mesma maneira que o RS232/UART, isto é, usando camadas, como mostra a Figura 2.8.

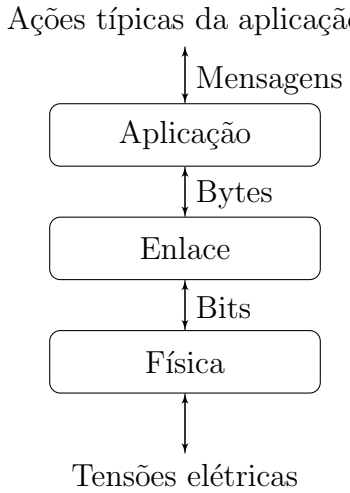

Figura 2.8: Camadas de protocolo na comunicação ponto-a-ponto que será implementada neste roteiro.

O protocolo I2C é um protocolo a dois fios – e, portanto, também é chamado de Two-Wire Interface, TWI[1]. Nele, há dois fios: SDA e SCL. SDA é o fio onde trafegam dados e SCL é o fio que carrega um sinal de clock.

TWI/I2C é um protocolo síncrono, isto é, um bit de dados só é transmitido quando acompanhado de uma borda num sinal de clock. Também, é um protocolo mestre-escravo. Isso significa

[1]O nome I2C é uma trademark da Philips, e por isso é evitado por outros fabricantes. Apesar disso, o protocolo não é uma patente fechada e pode ser usado livremente, desde que com outro nome.

que em um barramento há um dispositivo mestre e uma série de dispositivos escravos. O fluxo de dados em I2C é bi-direcional, mas a comunicação só pode ser iniciada por um dispositivo escravo caso tenha sido requisitada anteriormente pelo dispositivo mestre.

Quando o dispositivo mestre deseja iniciar o processo de comunicação, envia ao barramento o **endereço** do dispositivo que deseja contactar. O dispositivo escravo, então, fica ativo, aguardando comandos. O dispositivo mestre passa a enviar dados (bytes) que representam os comandos, tais quais implementados no dispositivo específico. Após, a comunicação é encerrada pelo dispositivo mestre, que volta o barramento para a posição parada (trata-se de uma condição de parada, ou stop condition).

Embora seja, teoricamente, possível implementar todo esse processo manualmente usando temporizadores e GPIO, microcontroladores modernos já trazem hardware e bibliotecas específicas que resolvem as questões de envio de endereços, de dados e condições de parada.

No caso do AT24C16, o endereço do dispositivo é configurado usando pinos específicos do dispositivo. Procure na datasheet quais são esses pinos.

Em termos computacionais, a EEPROM funciona como um grande vetor não-volátil de bytes. Assim, o acesso à EEPROM é completamente determinado pelas operações de escrita e leitura de dados. Portanto, é preciso implementar e testar as seguintes funções:

```
/* Escreve dado numa posicao da memoria */
void write_byte(int posicao, char dado);

/* Le dado encontrado numa posicao da memoria */
char read_byte(int posicao);
```

A implementação dessas funções depende da interpretação da datasheet e das bibliotecas disponíveis para sua plataforma microcontrolada.

2.6.1 Um Sistema de Arquivos

Uma vez que o problema de escrever e ler dados da memória não-volátil tenha sido resolvido, é preciso definir como esses dados serão organizados. Uma forma muito simples de organização, descrita nesta seção, é adequada para dispostivos tipo datalogger, que recebem dados sequencialmente e não precisam apagar dados acessados randomicamente. Na explicação desta solução, a notação $x[n]$ representa o n-ésimo byte da memória.

O sistema de arquivos se baseia em usar um byte no começo da memória – $x[0]$ – para armazenar o número de registros existentes na memória, assumindo que todos os registros têm o mesmo tamanho (em bytes). Assim, o procedimento para reiniciar o sistema de arquivos consiste em simplesmente escrever 0 nessa posição.

Ao receber uma instrução de escrever um novo registro de tamanho N bytes, o sistema grava o novo registro à partir da posição $N \times x[x[0]] + 1$ da memória. Esta posição – $N \times x[x[0]]$ – é o primeiro byte livre da memória. Após, incrementa $x[0]$.

Capítulo 3

Circuitos

3.1 Push Button e resistor de *pull*

Um dispositivo *push button* é uma chave mecânica que é ativada quando pressionada. Ao ser ativada, a chave fecha o contato entre dois pontos. Assim, pode ser conectado como mostra a Figura 3.1.

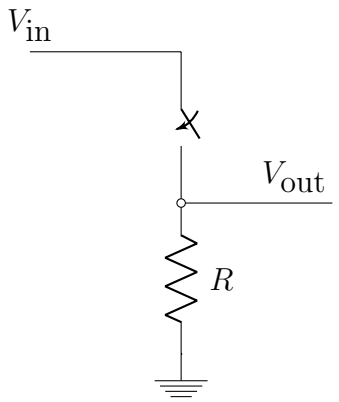

Figura 3.1: Circuito com uma chave.

No circuito, temos duas situações possíveis: a chave aberta e a chave fechada. Quando a chave está aberta, não passa corrente no circuito e, portanto, a tensão no resistor é zero. Quando a chave está fechada, o resistor entra em curto-circuito com a fonte, e, portanto, a tensão no dispositivo é V e a corrente no circuito é V/R.

Observe que, nessa estrutura, o resistor tem a função de induzir um estado-padrão no sistema, que opera quando a chave está aberta. Com a chave aberta, sem o resistor, V_{out}seria um nó flutuante, e, portanto, sujeito a oscilações de tensão devidas ao ruído eletromagnético do ambiente.

Na estrutura da Figura 3.1, o resistor faz com que o estado-padrão da saída do circuito seja baixo ($0V$), e, portanto, é chamado de resistor de *pull down* [1]. Se a chave e o resistor forem invertidos, o estado padrão do sistema passa a ser alto (V_{in}), e, assim, o resistor passa a ser chamado de resistor de *pull up*.

3.2 Potenciômetro

Um potenciômetro pode ser entendido como um resistor longo, no qual temos acesso a três terminais. Dois deles estão fixos nas pontas do resistor e o terceiro flutua entre eles, numa posição que é controlada pela rotação do botão. Como mostra a Figura 3.2, o potenciômetro pode ser conectado a uma fonte de tensão de forma a criar uma espécie de "divisor de tensão variável", cuja saída – a tensão V_{out}– se correlaciona à rotação do botão.

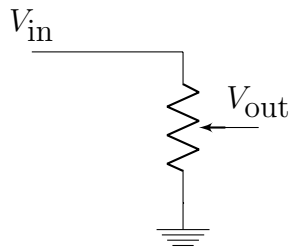

Figura 3.2: Circuito para acender um LED.

3.3 LED

O LED (*Light-Emitting Diode*) [2] é um diodo que emite luz na passagem de corrente. Um possível modelo para seu funcionamento elétrico é:

1. Se a tensão aplicada no LED é inferior a um limiar v_l, a corrente que passa por ele é nula;

2. Caso contrário, a corrente que passa pelo LED é tão grande quando possível e observa-se uma tensão de v_l nos terminais do LED.

A tensão v_l varia de acordo com a cor do LED e depende de características do material semicondutor do qual ele é feito.

Esse modelo significa que uma fonte de tensão conectada ao LED fornecerá corrente infinita. Isso rapidamente levará o LED (e, possivelmente, a própria fonte de tensão) a queimar. Por esse motivo, é preciso conectar o LED em série a um resistor, como mostra a Figura 3.3.

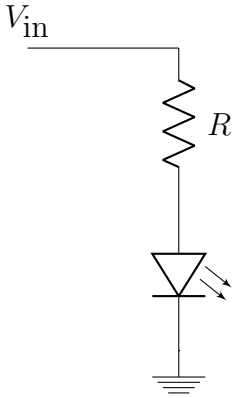

Figura 3.3: Circuito para acender um LED.

No circuito da Figura 3.3, observamos uma malha na qual passa uma corrente i. Assumindo que a tensão fornecida pela fonte é maior que o limiar de tensão para acender o LED ($V > v_l$), podemos verificar que a tensão no resistor é igual a (pela Lei de Kirchoff das Tensões) $V_r = V - V_l$. A Lei de Ohm nos permite inferir que a corrente na malha é igual a:

$$i = \frac{V - V_l}{R}. \tag{3.1}$$

Isso significa que é possível variar a corrente que passa pelo LED alterando a resistência do elemento colocado em série a ele. Uma corrente muito baixa levará a um brilho muito baixo, ao passo que uma corrente muito alta poderá danificar os dispositivos conectados.

3.4 Driver transistorizado

A montagem do LED seja bastante direta, e é possível trocar o LED diretamente por alguns outros componentes. Apesar disso, há situações em que o microcontrolador não é capaz de fornecer corrente suficiente para o funcionamento de algum dispositivo. Os circuitos tipo *driver* servem para garantir o fornecimento de uma corrente alta na sua saída, mesmo que a corrente na entrada seja baixa.

Este projeto de driver se baseia no Transistor Bipolar de Junção (TBJ). Trata-se de um dispositivo de três pólos – base, emissor e coletor – tal qual mostra a Figura 3.4.

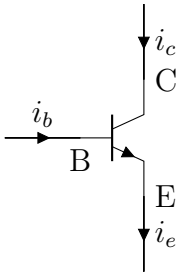

Figura 3.4: Transistor bipolar de junção, evidenciando a base (B), o emissor (E) e o coletor (C). Pela Lei de Kirchoff das correntes, $i_e = i_b + i_c$.

Uma propriedade importante do TBJ é que, quando a tensão v_c no coletor é alta e a tensão v_e no emissor é baixa, a corrente que entra pelo coletor passa a ser modulada pela corrente que entra pela base.

Quando a tensão v_{be} entre e base e emissor excede 0.7V, a junção base-emissor passa a se comportar como um diodo polarizado em sentido direto. Além disso, nessas condições, a corrente no coletor é igual à corrente da base multiplicada por um coeficiente β, que representa o *ganho de corrente em sentido direto* (na datasheet de um transistor, geralmente é chamado de *forward current gain* ou *DC current gain*):

$$i_c = \beta i_b. \tag{3.2}$$

Usando essa propriedade, podemos projetar um circuito no qual o microcontrolador fornece

tensão na base do transistor. Como mostra a Figura 3.5, um resistor limitará a corrente fornecida e o transistor criará as condições para o ganho de corrente. Trata-se de uma topologia bem conhecida, na qual o transitor atua como uma chave controlada pela tensão aplicada na base.

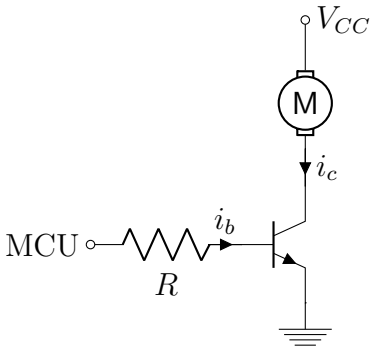

Figura 3.5: Circuito driver.

Na topologia da Figura 3.5, temos dois casos possíveis. Quando o sinal do MCU está em 0V, a tensão na base do transistor também se torna 0V, o que faz com que o transistor atue como uma chave aberta. Porém, quando o sinal do MCU está em 3.3V ou 5V, induzimos corrente na base – e, portanto, no pino do microcontrolador – dada por:

$$i_b = \frac{V_{\text{CC}} - 0.7}{R} = \frac{3.3 - 0.7}{R} = \frac{2.4}{R}. \tag{3.3}$$

Desta forma, temos que a corrente no coletor, e, portanto, no motor, é dada por:

$$i_c = \beta i_b = \beta \frac{2.4}{R}. \tag{3.4}$$

O cálculo da corrente i_c não considera que a força contra-eletromotriz do motor se modifica. Se i_c for maior que a corrente requerida pelo motor, a tensão no motor aumenta (devido a força contra-eletromotriz), causando uma consequente queda na tensão no emissor. Daí, o transistor passa a operar em modo de saturação e pode ser entendido como uma chave fechada, com queda de tensão entre o coletor e o emissor de aproximadamente 1V.

Assim, podemos usar circuitos transistorizados para controlar um motor de 5V usando um microcontrolador que opera em 3.3V. Além disso, controlamos um dispositivo que consome muita corrente apenas fornecendo uma corrente baixa. A corrente "adicional" que passa pelo motor é fornecida diretamente por uma fonte DC externa ao microcontrolador.

Referências Bibliográficas

[1] Wikipedia. Pull-up resistors. https://en.wikipedia.org/wiki/Pull-up_resistor.

[2] Wikipedia. Light-emitting diodes. https://en.wikipedia.org/wiki/Light-emitting_diode.