5. PORTAS DE ENTRADA E SAÍDA (I/Os)

A primeira dúvida ao se trabalhar com um microcontrolador é como funcionam os seus pinos, ou seja, com são escritos e lidos dados nas suas entradas e saídas. Neste capítulo, é feita uma introdução à programação do ATmega328, começando com os conceitos de mais fácil compreensão: a leitura e escrita de seus pinos de I/O. O estudo inicia com o acionamento de LEDs e a leitura de botões, seguido das técnicas para o uso de displays de 7 segmentos e de cristal líquido (LCD 16×2). Também se explica o emprego de rotinas de atraso, comuns na programação microcontrolada.

O ATmega328 possui 3 conjuntos de pinos de entrada e saída (I/Os): PORTB, PORTC e PORTD; respectivamente, pinos PB7 .. PB0, PC6 .. PC0 e PD7 .. PD0, todos com a função Lê – Modifica – Escreve. Isso significa que a direção de um pino pode ser alterada sem mudar a direção de qualquer outro pino do mesmo PORT (instruções SBI e CBI[19]). Da mesma forma, os valores lógicos dos pinos podem ser alterados individualmente, bem como a habilitação dos resistores de *pull-up* para os pinos configurados como entrada. Cada PORT possui um registrador de saída com características simétricas, isto é, com a mesma capacidade para drenar ou suprir corrente, suficiente para alimentar LEDs diretamente (20 mA por pino). O cuidado a se ter é respeitar a máxima corrente total que o componente e que cada PORT suporta, 200 mA e 100 mA, respectivamente[20]. Todas os pinos têm resistores de *pull-up* internamente e diodos de proteção entre o VCC e o terra, além de uma capacitância de 10 pF, como indicado na fig. 5.1.

[19] SBI = ativa um bit (coloca em 1). CBI = limpa um bit (coloca em 0).

[20] Sempre que possível deve-se utilizar a menor corrente. É importante consultar o manual do fabricante para a observação das características elétricas do microcontrolador.

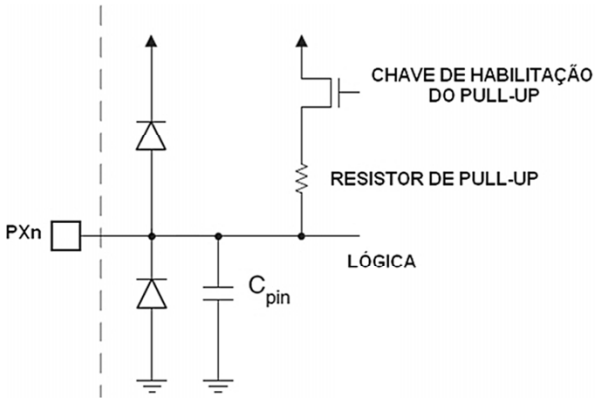


Fig. 5.1 – Esquema geral dos pinos de I/O (PXn) do ATmega.

Os registradores responsáveis pelos pinos de I/O são:

- **PORTx**: registrador de dados, usado para escrever nos pinos do PORTx.
- **DDRx**: registrador de direção, usado para definir se os pinos do PORTx são entrada ou saída.
- **PINx**: registrador de entrada, usado para ler o conteúdo dos pinos do PORTx.

Em resumo, para o uso de um pino de I/O, deve-se primeiro definir se ele será entrada ou saída escrevendo-se no registrador DDRx. Então, a escrita no registrador PORTx alterará o estado lógico do pino se ele for saída, ou poderá habilitar o *pull-up* interno, se ele for entrada[21]. Os estados lógicos dos pinos do PORT são lidos do registrador PINx. É importante notar que para a leitura do PINx logo após uma escrita no PORTx ou DDRx, deve ser gasto pelo menos um ciclo de máquina para a sincronização dos dados pelo microcontrolador.

[21] Se algum pino não for utilizado é recomendado que o seu nível lógico seja definido. Entradas com nível flutuante devem ser evitadas para evitar o consumo de corrente quando o pino não estiver sendo empregado. Neste caso a Atmel recomenda a habilitação do *pull-up* (para maiores detalhes ver o manual do fabricante).

Na tab. 5.1, é apresentada as configurações dos bits de controle dos registradores responsáveis pela definição do comportamento dos pinos (ver a tab. 2.1 – registradores de entrada e saída do ATmega328).

Tab. 5.1 - Bits de controle dos pinos dos PORTs.

DDXn*	PORTXn	PUD (no MCUCR)	I/O	Pull-up	Comentário
0	0	x	Entrada	Não	Alta impedância (Hi-Z).
0	1	0	Entrada	Sim	PXn irá fornecer corrente se externamente for colocado em nível lógico 0.
0	1	1	Entrada	Não	Alta impedância (Hi-Z).
1	0	x	Saída	Não	Saída em zero (drena corrente).
1	1	x	Saída	Não	Saída em nível alto (fornece corrente).

*X = B, C ou D; n = 0, 1, ... ou 7.

Quando o bit PUD (*Pull-Up Disable*) no registrador MCUCR (MCU *Control Register*) está em 1 lógico, os *pull-ups* em todos os PORTs são desabilitados, mesmo que os bits DDXn e PORTXn estejam configurados para habilitá-los.

5.1 ROTINAS SIMPLES DE ATRASO

Rotinas de atraso são muito comuns na programação de microcontroladores. São realizadas fazendo-se a CPU gastar ciclos de máquina na repetição de instruções. Para se calcular o exato número de ciclos de máquina gastos em uma sub-rotina de atraso é necessário saber quantos ciclos cada instrução consome. Para exemplificar é utilizado o programa *assembly* a seguir.

```
Atraso:

    DEC  R3        //decrementa R3, começa com o valor 0x00
    BRNE Atraso    //enquanto R3 > 0 fica decrementando R3, desvio para Atraso
    DEC  R2        //decrementa R2, começa com o valor 0x00
    BRNE Atraso    //enquanto R2 > 0 volta a decrementar R3
    RET            //retorno da sub-rotina
```

No programa acima, a instrução DEC consome 1 ciclo, a instrução BRNE consome 2 ciclos e na última vez, quando não desvia mais, consome 1 ciclo. Como os registradores R3 e R2 possuem o valor zero inicialmente[22] (o primeiro decremento os leva ao valor 255) e o decremento de R3 é repetido dentro do laço de R2, espera-se que haja 256 decrementos de R3 vezes 256 decrementos de R2. Se for considerado 3 ciclos para DEC e BRNE, tem-se aproximadamente $(3 \times 256 \times 256) + (3 \times 256)$ ciclos, ou seja, 197.376 ciclos. A parcela 3×256, que é o tempo gasto no decremento de R2, pode ser desprezada, pois afeta pouco o resultado final. Assim, a fórmula aproximada seria dada por $3 \times 256 \times 256$, igual a 196.608 ciclos.

Entretanto, BRNE não consome dois ciclos no último decremento e sim um. Desta forma, o cálculo preciso é mais complexo:

$$\Big(\big((3 \text{ ciclos} \times 255) + 2 \text{ ciclos}\big) + 3 \text{ ciclos}\Big) \times 255 + 769 \text{ ciclos} = 197.119 \text{ ciclos.}$$

Para melhor compreensão, observar a fig. 5.2. Se forem considerados os ciclos gastos para a chamada da sub-rotina, com a instrução RCALL (3 ciclos) e os ciclos para o retorno, com a instrução RET (4 ciclos), tem-se um gasto total da chamada da sub-rotina até seu retorno de 197.126 ciclos.

Atraso:
DEC R3
BRNE Atraso $\left[\begin{matrix}1 \text{ ciclo} \\ 2 \text{ ciclos}\end{matrix}\right] \times 255 + \left[\begin{matrix}1 \text{ ciclo} \\ 1 \text{ ciclo}\end{matrix}\right] = \left[\begin{matrix}767 \text{ ciclos} \\ + \\ 1 \text{ ciclo} \\ + \\ 2 \text{ ciclos}\end{matrix}\right] \times 255 + \left[\begin{matrix}767 \text{ ciclos} \\ + \\ 1 \text{ ciclo} \\ + \\ 1 \text{ ciclo}\end{matrix}\right] = 197119 \text{ ciclos}$
DEC R2
BRNE Atraso

Fig. 5.2 – Cálculo preciso da sub-rotina de atraso.

O tempo gasto pelo microcontrolador dependerá da frequência de trabalho utilizada. Como no AVR um ciclo de máquina equivale ao inverso da frequência do *clock* (período), o tempo gasto será dado por:

[22] Após a inicialização do ATmega os valores dos registradores R0:R31 são indeterminados, considera-se zero para a simplificação do programa. Caso seja necessário que os registradores tenham um valor conhecido, eles devem ser inicializados. Uma vez executada a sub-rotina de atraso e considerando-se que os registradores empregados nela não são usados em outras partes do programa, eles sempre estarão inicialmente em zero. Na dúvida, os registradores sempre devem ser inicializados.

$$Tempo\ Gasto = N°\,de\ ciclos \times \frac{1}{Freq.\ de\ trabalho} \quad (5.1)$$

Logo, para o exemplo acima, com um *clock* de 16 MHz (período de 62,5 ns), da chamada da sub-rotina até seu retorno, resulta em:

$$Tempo\ Gasto = 197.126 \times 62{,}5\ \text{ns} = 12{,}32\ \text{ms}$$

Exercícios:

5.1 – Qual o tempo aproximado e exato gasto pelo ATmega para a execução da sub-rotina abaixo?

```
ATRASO:
        LDI  R19,X     //carrega R19 com o valor X (dado abaixo)
   volta:
        DEC  R18       //decrementa R18, começa com 0x00
        BRNE volta     //enquanto R18 > 0 volta a decrementar R18
        DEC  R19       //decrementa R19
        BRNE volta     //enquanto R19 > 0 vai para volta
        RET
```

a) X = 0, f_{clk} = 16MHz.

b) X = 10, f_{clk} = 1MHz.

5.2 – Qual o tempo aproximado gasto para a execução da sub-rotina abaixo para uma frequência de operação do ATmega de 8 MHz (fluxograma na fig. 4.4b)?

```
ATRASO:
        LDI  R19,0x02
    volta:
        DEC  R17       //decrementa R17, começa com 0x00
        BRNE volta     //enquanto R17 > 0 fica decrementando R17
        DEC  R18       //decrementa R18, começa com 0x00
        BRNE volta     //enquanto R18 > 0 volta a decrementar R18
        DEC  R19       //decrementa R19
        BRNE volta     //enquanto R19 > 0 vai para volta
        RET
```

5.3 – Desenvolva uma sub-rotina em *assembly* para produzir um atraso de aproximadamente 0,5 s para o ATmega operando a 16 MHz.

5.4 – Para o programa abaixo, escrito na linguagem C, qual é o tempo aproximado gasto pelo ATmega para a sua execução (f_{clk} = 20 MHz). Considere que são gastos 3 ciclos de *clock* para que uma repetição do laço **for** seja realizada.

```
unsigned int i, j;

for(i=256; i!=0; i--)
{
     for(j=65535; j!=0; j--);
}
```

5.2 LIGANDO UM LED

Para começar o trabalho com o ATmega, será feito o acionamento de um LED. Como o Arduino possui um LED ligado diretamente ao pino PB5 do ATmega, não é necessário o uso de um circuito adicional (fig. 5.3).

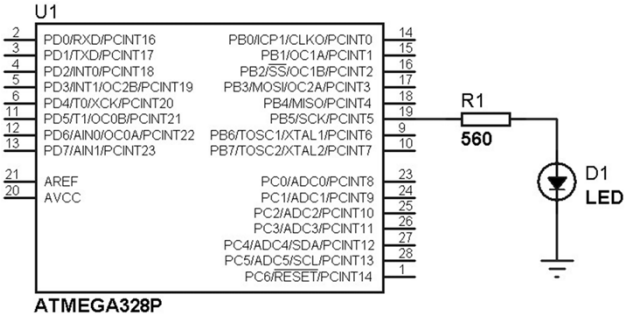


Fig. 5.3 – Ligando um LED.

Como o microcontrolador trabalha de acordo com um programa, por mais simples que pareça ligar um LED, existe uma infinidade de possibilidades: o LED pode ser piscado, a frequência pode ser alterada, o número de vezes que se liga e desliga pode ser ajustada, bem como o tempo de acionamento.

Quando se aprende a programar um microcontrolador, o primeiro programa é piscar um LED, tal como o "*Hello Word!*" quando se programa um computador. Para isso basta ligar o LED, esperar um tempo, desligar o

LED, esperar um tempo e voltar novamente a ligá-lo, repetindo o processo indefinidamente. O programa em *assembly* para o Arduino que executa essa tarefa é dado a seguir (o fluxograma do programa pode ser visto na fig. 4.4a).

Pisca_LED.asm

```
//---------------------------------------------------------------------
.equ LED   = PB5       //LED é o substituto de PB5 na programação
.ORG 0x000             //endereço de início de escrita do código

INICIO:
    LDI R16,0xFF       //carrega R16 com o valor 0xFF
    OUT DDRB,R16       //configura todos os pinos do PORTB como saída
PRINCIPAL:
    SBI PORTB, LED     //coloca o pino PB5 em 5V
    RCALL ATRASO       //chama a sub-rotina de atraso
    CBI PORTB, LED     //coloca o pino PB5 em 0V
    RCALL ATRASO       //chama a sub-rotina de atraso
    RJMP PRINCIPAL     //volta para PRINCIPAL

ATRASO:                //atraso de aprox. 200 ms
    LDI R19,16
 volta:
    DEC  R17           //decrementa R17, começa com 0x00
    BRNE volta         //enquanto R17 > 0 fica decrementando R17
    DEC  R18           //decrementa R18, começa com 0x00
    BRNE volta         //enquanto R18 > 0 volta a decrementar R18
    DEC  R19           //decrementa R19
    BRNE volta         //enquanto R19 > 0 vai para volta
    RET
//---------------------------------------------------------------------
```

O programa *assembly* consumiu 30 bytes de memória de programa (*flash*) com o uso de 15 instruções, todas de 16 bits.

O programa escrito na linguagem C é apresentado a seguir. É importante no AVR *Studio* configurar a compilação para a otimização máxima, atalho <Alt + F7>, ver a fig. 3.12. Após a montagem[23], o programa resultou em 178 bytes de memória de programa (as vantagens e desvantagens do *assembly* sobre o C e vice-versa foram apresentadas no capítulo 4).

[23] Ver nota de rodapé de número 12 na página 46.

Pisca_LED.c

```
//---------------------------------------------------------------------------
#define F_CPU 16000000UL   /*define a frequência do microcontrolador 16MHz
                              (necessário para usar as rotinas de atraso)*/
#include <avr/io.h>        //definições do componente especificado
#include <util/delay.h>    /*biblioteca para o uso das rotinas de
                                                  _delay_ms() e _delay_us()*/

//Definições de macros - empregadas para o trabalho com os bits
#define  set_bit(Y,bit_x) (Y|=(1<<bit_x))    /*ativa o bit x da
                                                  variável Y (coloca em 1)*/
#define  clr_bit(Y,bit_x) (Y&=~(1<<bit_x))   /*limpa o bit x da variável Y
                                                    (coloca em 0)*/
#define  tst_bit(Y,bit_x) (Y&(1<<bit_x))     /*testa o bit x da variável Y
                                                 (retorna 0 ou 1)*/
#define  cpl_bit(Y,bit_x) (Y^=(1<<bit_x))    /*troca o estado do bit x da
                                                  variável Y (complementa)*/
#define LED PB5   //LED é o substituto de PB5 na programação

int main( )
{
    DDRB = 0xFF;  //configura todos os pinos do PORTB como saídas

    while(1)                  //laço infinito
    {
         set_bit(PORTB,LED);  //liga LED
         _delay_ms(200);      //atraso de 200 ms
         clr_bit(PORTB,LED);  //desliga LED
         _delay_ms(200);      //atraso de 200 ms
    }
}
//---------------------------------------------------------------------------
```

Para compreender o programa acima, é importante dominar o trabalho com bits (ver a seção 4.5.9). Esse conhecimento é fundamental para se programar eficientemente o ATmega.

O acionamento de um LED é uma das tarefa mais simples que se pode realizar com um microcontrolador. LEDs sinalizadores estão presentes em quase todos os projeto eletrônicos e pode-se dar muito maior complexidade a função de sinalização, como por exemplo, indicar a carga de processamento da CPU, aumentando-se ou diminuindo-se a frequência com que um LED é ligado e desligado.

Exercícios:

5.5 – Faça um programa em *assembly* para piscar um LED a cada 1 s.

5.6 – Utilizando a macro para complementar um bit (cpl_bit), faça um programa para piscar um LED a cada 500 ms.

5.7 – Desenvolva um programa para piscar um LED rapidamente 3 vezes e 3 vezes lentamente.

5.8 – Utilizando o deslocamento de bits crie um programa em *assembly* que ligue 8 LEDs[24] (ver a fig. 5.4a), da seguinte forma:

a) Ligue sequencialmente 1 LED da direita para a esquerda (o LED deve permanecer ligado até que todos os 8 estejam ligados, depois eles devem ser desligados e o processo repetido).

b) Ligue sequencialmente 1 LED da esquerda para a direita, mesma lógica da letra a.

c) Ligue sequencialmente 1 LED da direita para a esquerda, desta vez somente um LED deve ser ligado por vez.

d) Ligue sequencialmente 1 LED da esquerda para a direita e vice-versa (vai e volta), só um LED deve ser ligado por vez.

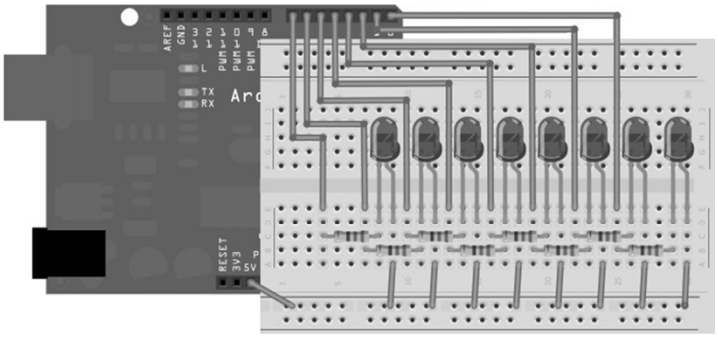
e) Ligue todos os LEDs e apague somente 1 LED de cada vez, da direita para a esquerda e vice-versa (vai e volta), somente um LED deve ser apagado por vez.

f) Mostre uma contagem binária crescente (0-255) com passo de 250 ms.

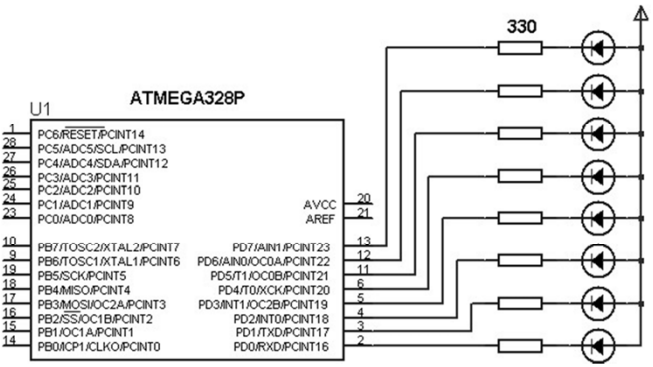
g) Mostre uma contagem binária decrescente (255-0) com passo de 250 ms.

[24] Como o Arduino utiliza um programa de *boot loader*, ele emprega os pinos TXD e RXD para a gravação da memória de programa (pinos PD0 e PD1). Quando não se utiliza a IDE do Arduino para a gravação, esse pinos devem ser configurados explicitamente pelo programador para serem pinos de I/O genéricos, caso contrário os pinos não se comportarão como esperado (configuração *default*). Em C é necessário acrescentar a seguinte linha de código: **UCSR0B = 0x00;** //desabilita RXD e TXD.

Importante: quando se grava o Arduino, os pinos 0 e 1 (PD0 e PD1) não devem estar conectados a nenhum circuito, caso contrário, poderá haver erro de gravação. Se algum *shield* utilizar os referidos pinos, é aconselhável gravar primeiro o Arduino e somente depois conectá-lo.



a)



b)

Fig. 5.4 – Sequencial com 8 LEDs: a) montagem para o Arduino e b) esquemático.

5.3 LENDO UM BOTÃO (CHAVE TÁCTIL)

Quando se começa o estudo de um microcontrolador, além de se ligar um LED, um dos primeiros programas é liga-lo ao se pressionar um botão (tecla ou chave táctil). O problema é que na prática, botões apresentam o chamado *bounce,* um ruído que pode ocorrer ao se pressionar ou soltar o botão. Esse ruído produz uma oscilação na tensão proveniente do botão, ocasionando sinais lógicos aleatórios que podem produzir leituras errôneas. Se o ruído existir, geralmente ele desaparece após

aproximadamente 10 ms. Esse tempo depende das características físicas e elétricas do botão e do circuito onde ele se encontra.

Dependendo da configuração do circuito do botão, o seu pressionar pode resultar em um sinal lógico alto ou baixo. Isso depende da forma como se emprega o resistor necessário para a leitura do botão: com *pull-up*, entre a alimentação (VCC) e o botão, ou com *pull-down*, entre o botão e o terra. Na fig. 5.5, é exemplificado o ruído que pode ser produzido por um botão com essas configurações.

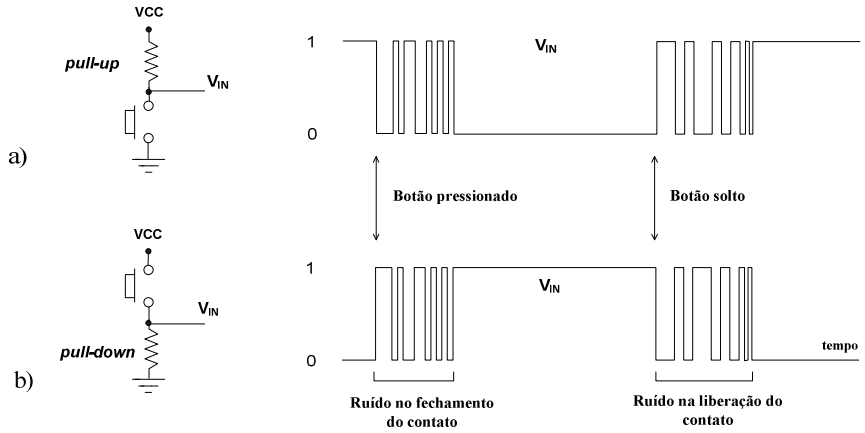


Fig. 5.5 – Exemplo do ruído que pode ser gerado ao se pressionar e soltar um botão: a) usando um resistor de *pull-up* e b) usando um resistor de *pull-down*.

O ruído no fechamento do contato, geralmente não produz problemas, pois muitas vezes não existe ou é muito pequeno para ser notado. O ruído na liberação do contato é comum e possui maior duração devido, principalmente, ao contato mecânico do botão. Quando se pressiona o botão, os seus contatos são mantidos pela pressão aplicada sobre ele; quando ele é solto, o contato se desfaz e existe um repique mecânico. Além disso, o circuito do botão possui intrinsicamente indutâncias e capacitâncias que geram ruídos elétricos quando o circuito é fechado ou aberto.

Na fig. 5.6, é apresentado o ruído obtido experimentalmente quando um botão (comum em eletrônica) com um *pull-up* de 10 kΩ e ligado a 5 V é liberado (avaliado com o emprego de um osciloscópio). Pode-se notar que o ruído tem duração de aproximadamente 3 ms (divisão horizontal de 50 μs e vertical de 1 V).

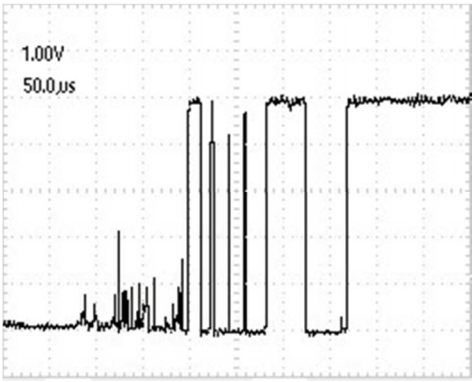


Fig. 5.6 – Ruído real gerado ao se soltar um botão com *pull-up*.

Observando-se o ruído da fig. 5.6 e sabendo-se que a velocidade de operação de um sistema digital depende da sua frequência de trabalho, quanto maior a frequência de leitura do botão mais sensível ao ruído o sistema se torna. Para resolver esse problema é necessário o uso de capacitores ou outros componentes eletrônicos. Todavia, ele é facilmente resolvido quando o sistema é controlado por um programa.

A técnica de hardware para eliminação do *bounce* é o chamado *debounce*. Em sistemas microcontrolados, o *debounce* é feito via software, sem a necessidade de componentes externos para a filtragem do ruído.

A seguir, é apresentado o fluxograma para um programa que troca o estado de um LED toda vez que um botão é pressionado (fig. 5.7). O mesmo emprega um pequeno tempo para eliminar o eventual ruído do botão quando solto. O ruído no pressionamento não é considerado. O circuito empregado é apresentado na fig. 5.8. Como o ATmega possui resistores

internos de *pull-up*, para a leitura de um botão basta somente ligá-lo diretamente ao terra e a um dos pinos de I/O do microcontrolador.

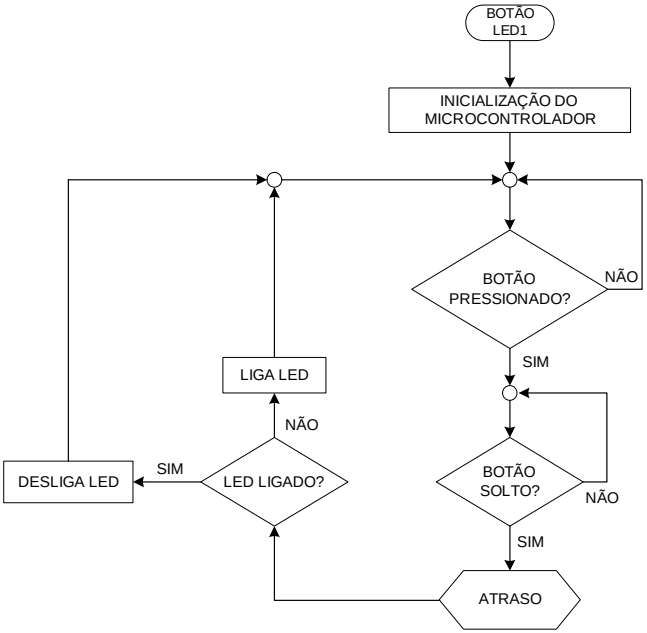
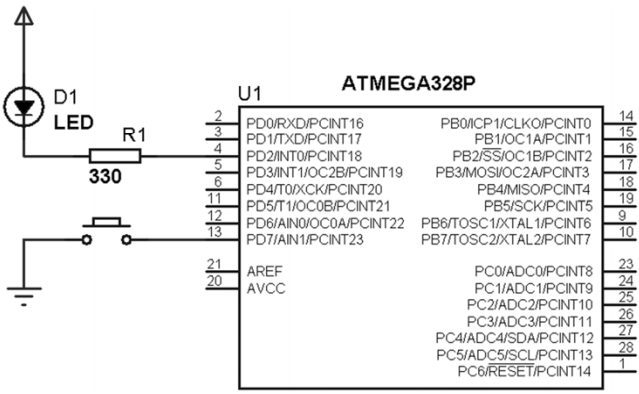
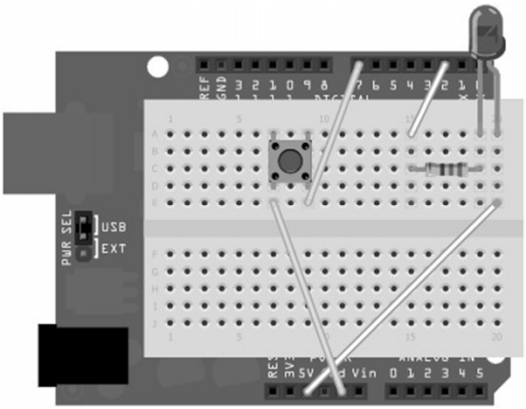


Fig. 5.7 – Fluxograma do programa para ligar e apagar um LED com um botão.



a)



b)

Fig. 5.8 – Circuito para ligar e apagar um LED com um botão: a) esquemático e b) montagem para o Arduino.

O programa em *assembly* é apresentado a seguir (ver o apêndice A para detalhes das instruções *assembly* do ATmega328). É bom compará-lo com o fluxograma da fig. 5.7 para sua compreensão. Também é importante observar que algumas instruções em *assembly* só trabalham com alguns dos 32 registradores de propósito geral. Após a montagem, o tamanho do código foi de 42 bytes (21 instruções).

Botao_LED.asm

```
//============================================================================== //
//      LIGANDO E DESLIGANDO UM LED QUANDO UM BOTÃO É PRESSIONADO                 //
//============================================================================== //
//DEFINIÇÕES
.equ LED   = PD2  //LED é o substituto de PD2 na programação
.equ BOTAO = PD7  //BOTAO é o substituto de PD7 na programação
.def AUX   = R16  /*R16 tem agora o nome de AUX (nem todos os 32 registradores
                        de uso geral podem ser empregados em todas as instruções) */
//------------------------------------------------------------------------------
.ORG 0x000      /*endereço de início de escrita do código na memória flash,
                              após o reset o contador do programa aponta para cá.*/
Inicializacoes:

     LDI  AUX,0b00000100  //carrega AUX com o valor 0x04 (1 = saída e 0 = entrada)
     OUT  DDRD,AUX        //configura PORTD, PD2 saída e demais pinos entradas
     LDI  AUX,0b11111111  /*habilita o pull-up para o botão e apaga o LED (pull-up em
                                                              todas as entradas)*/
     OUT  PORTD,AUX

     NOP           /*sincronização dos dados do PORT. Necessário somente para
                              uma leitura imediatamente após uma escrita no PORT*/
```

```
//-------------------------------------------------------------------------------
//LAÇO PRINCIPAL
//-------------------------------------------------------------------------------
Principal:
     SBIC  PIND,BOTAO     //verifica se o botão foi pressionado, senão
     RJMP  Principal      //volta e fica preso no laço Principal

Esp_Soltar:
     SBIS  PIND,BOTAO     //se o botão não foi solto, espera soltar
     RJMP  Esp_Soltar
     RCALL Atraso         /*após o botão ser solto gasta um tempo para eliminar o
                                                        ruído proveniente do mesmo*/
     SBIC  PORTD,LED      //se o LED estiver apagado, liga e vice-versa
     RJMP  Liga
     SBI   PORTD,LED      //apaga o LED
     RJMP  Principal      //volta ler botão
Liga:
     CBI   PORTD,LED      //liga LED
     RJMP  Principal      //volta ler botão
//-------------------------------------------------------------------------------
//SUB-ROTINA DE ATRASO - Aprox. 12 ms a 16 MHz
//-------------------------------------------------------------------------------
Atraso:
     DEC  R3              //decrementa R3, começa com 0x00
     BRNE Atraso          //enquanto R3 > 0 fica decrementando R3
     DEC  R2
     BRNE Atraso          //enquanto R2 > 0 volta a decrementar R3
     RET
//===============================================================================
```

Muitos microcontroladores da família AVR necessitam que o *Stack Pointer* seja inicializado pelo programa em *assembly* (em C a inicialização é feita automaticamente pelo compilador). Para o ATmega328 isso não é necessário, mas caso desejado, ela poderia ser feita com:

```
//inicialização do Stack Pointer deve ser feita com o endereço final da SRAM
LDI  R16, high(RAMEND)
OUT  SPH, R16             //registrador SPH = parte alta do endereço
LDI  R16, low(RAMEND)
OUT  SPL, R16             //registrador SPL = parte baixa do endereço
```

Para comparação com o programa *assembly*, é apresentado a seguir o código em C com a mesma funcionalidade (fluxograma da fig. 5.7).

Botao_LED.c

```
//=============================================================================== //
//    LIGANDO E DESLIGANDO UM LED QUANDO UM BOTÃO É PRESSIONADO                    //
//=============================================================================== //
#define F_CPU 16000000UL  /*define a frequência do microcontrolador 16MHz (necessário
                                                    para usar as rotinas de atraso)*/
#include <avr/io.h>       //definições do componente especificado
#include <util/delay.h>   //bibliot. para as rotinas de _delay_ms() e delay_us()

//Definições de macros - para o trabalho com os bits de uma variável
#define set_bit(Y,bit_x)(Y|=(1<<bit_x))  //ativa o bit x da variável Y (coloca em 1)
#define clr_bit(Y,bit_x)(Y&=~(1<<bit_x)) //limpa o bit x da variável Y (coloca em 0)
#define cpl_bit(Y,bit_x)(Y^=(1<<bit_x))  //troca o estado do bit x da variável Y
#define tst_bit(Y,bit_x)(Y&(1<<bit_x))   //testa o bit x da variável Y (retorna 0 ou 1)

#define LED   PD2   //LED é o substituto de PD2 na programação
#define BOTAO PD7   //BOTAO é o substituto de PD7 na programação
//-----------------------------------------------------------------------------------
int main()
{
     DDRD = 0b00000100;   //configura o PORTD, PD2 saída, os demais pinos entradas
     PORTD= 0b11111111;   /*habilita o pull-up para o botão e apaga o LED (todas as
                                                  entradas com pull-ups habilitados)*/
     while(1) //laço infinito
     {
          if(!tst_bit(PIND,BOTAO))//se o botão for pressionado executa o if
          {
               while(!tst_bit(PIND,BOTAO));   //fica preso até soltar o botão

               _delay_ms(10);  //atraso de 10 ms para eliminar o ruído do botão

               if(tst_bit(PORTD,LED))    //se o LED estiver apagado, liga o LED
                    clr_bit(PORTD,LED);
               else                      //se não apaga o LED
                    set_bit(PORTD,LED);

               //o comando cpl_bit(PORTD,LED) pode substituir este laço if-else

          }//if do botão pressionado

     }//laço infinito
}
//=============================================================================
```

Outra variante da leitura de botões, é executar o que necessita ser feito imediatamente após o botão ser pressionado e avaliar se o botão já foi solto. Após isso, então, se gasta um pequeno tempo para eliminar o *bounce*. O fluxograma da fig. 5.9 apresenta essa ideia.

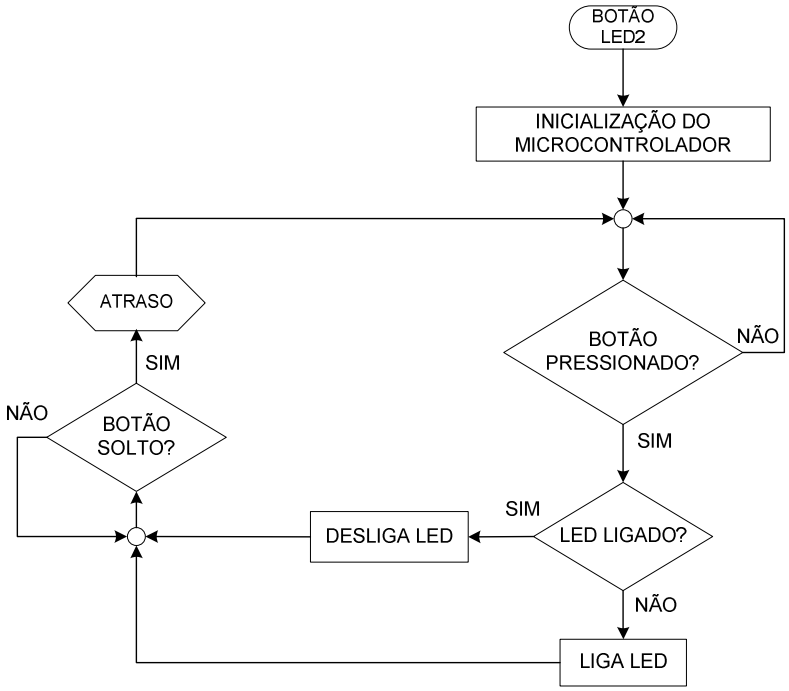


Fig. 5.9 – Fluxograma para ligar ou desligar imediatamente um LED após um botão ser pressionado.

Quando se necessita executar repetidamente uma determinada ação, poderia ser o incremento de uma variável, por exemplo, outra maneira de leitura de um botão deve ser empregada. Nesse caso, a leitura do botão ficará sempre dentro de um laço com uma rotina de tempo adequada para a realização da tarefa. O ruído é eliminado pelo tempo gasto. O fluxograma da fig. 5.10 apresenta essa ideia utilizada para piscar um LED enquanto um botão é mantido pressionado.

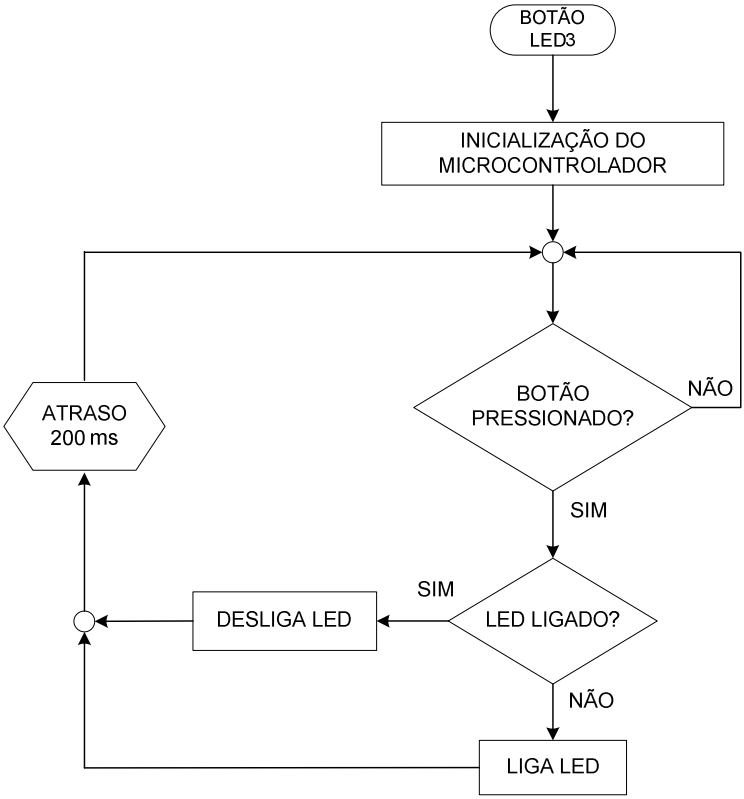


Fig. 5.10 – Fluxograma para piscar um LED enquanto um botão é mantido pressionado.

Em resumo, existem três formas para o *debounce* por software:

1. Após o botão ser pressionado, esperar o mesmo ser solto e gastar um pequeno tempo para que o ruído desapareça e, então, efetuar a ação correspondente.
2. Após o botão ser pressionado, efetuar imediatamente a ação correspondente, esperar o botão ser solto e depois esperar um pequeno tempo.
3. Após o botão ser pressionado, efetuar imediatamente a ação correspondente e depois esperar um tempo adequado para repetir a leitura do botão.

Se houver ruído quando o botão é pressionado, um atraso para eliminá-lo pode ser necessário. Todavia, a lógica para o *debounce* não muda. Dependendo da ação que será feita ao se pressionar o botão, gastar um determinado tempo para o *debounce* pode ser desnecessário. Isso dependerá somente do tempo que a ação leva para ser executada e do tempo para uma nova leitura do botão.

Exercícios:

5.9 – Elaborar um programa para ligar imediatamente um LED após o pressionar de um botão, com uma rotina de atraso de 10 ms para eliminação do *bounce*.

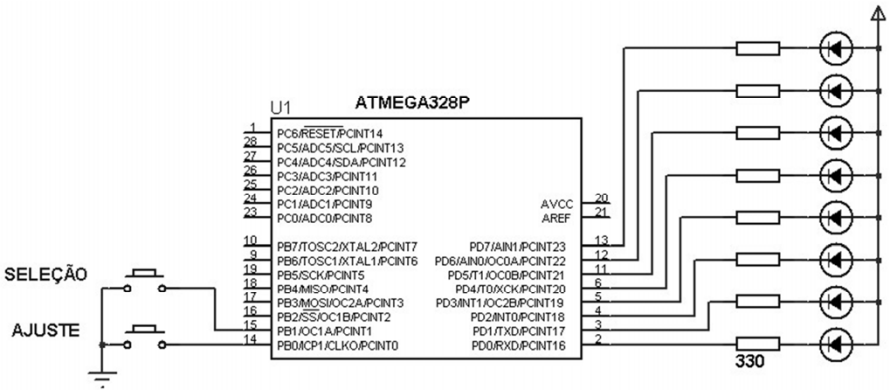
5.10 – Elaborar um programa que troque o estado do LED se o botão continuar sendo pressionado. Utilize uma frequência que torne agradável o piscar do LED.

5.11 – Elaborar um programa para aumentar a frequência em que um LED liga e desliga, enquanto um botão estiver sendo pressionado, até o momento em que o LED ficará continuamente ligado. Quando o botão é solto o LED deve ser desligado.

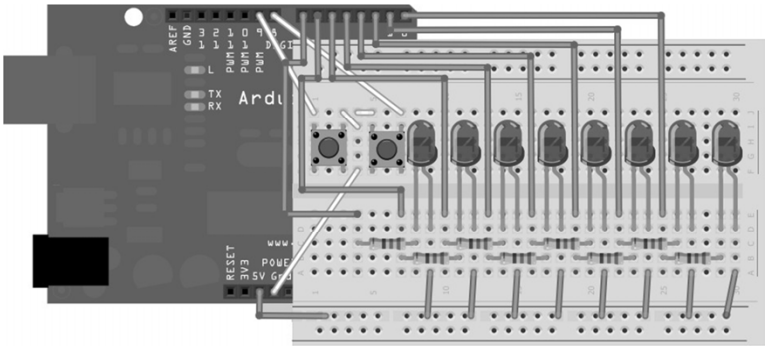
Qual a aplicação prática dessa técnica, imaginando que o botão pode ser o sinal proveniente de algum sensor e o LED algum dispositivo sinalizador?

5.12 – No exercício 5.8, foram propostas 7 animações com 8 LEDs. Crie outra animação, totalizando 8. Depois empregue dois botões: um será o AJUSTE, que quando pressionado permitirá que o outro botão (SELEÇÃO) selecione a função desejada, ver a fig. 5.11. Cada vez que o botão SELEÇÃO for pressionado, um dos oito LEDs deverá acender para indicar a função escolhida; exemplo: 00000100 => LED 3 ligado, função 3 selecionada. Quando o botão de AJUSTE for solto, o sistema começa a funcionar conforme a função escolhida.

Desenvolva a programação por partes, unindo-as e testando com cuidado. Não esqueça: um bom programa é bem comentado e organizado!



a)



b)

Fig. 5.11 – Sequencial de LEDs: a) esquemático e b) montagem no Arduino.

5.4 ACIONANDO DISPLAYS DE 7 SEGMENTOS

Um componente muito empregado no desenvolvimento de sistemas microcontrolados é o *display* de 7 segmentos. Esses *displays* geralmente são compostos por LEDs arranjados adequadamente em um encapsulamento, produzindo os dígitos numéricos que lhe são característicos. Na fig. 5.12, é apresentado o diagrama esquemático para uso de um *display* de anodo comum e os caracteres mais comuns produzidos por esse. Nos *displays* com catodo comum, o ponto comum dos LEDs é o terra e a tensão de alimentação deve ser aplicada individualmente em cada LED do *display*.

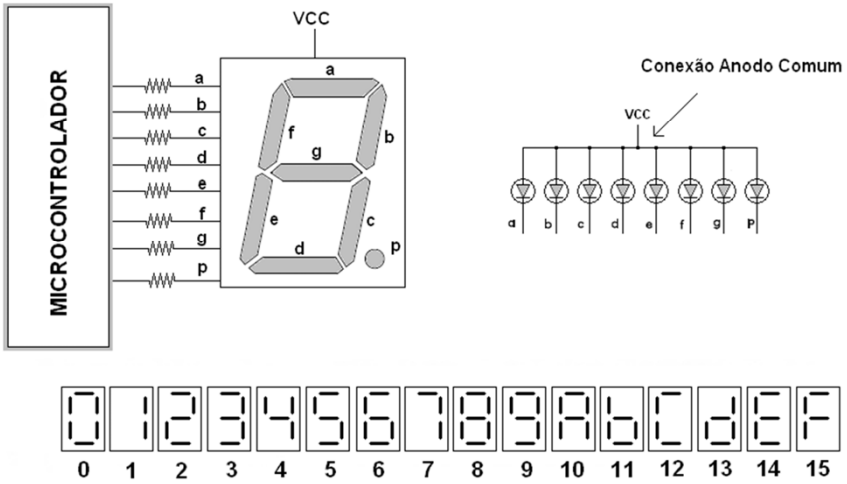


Fig. 5.12 – *Display* de 7 segmentos anodo comum.

Para o emprego de *displays*, é necessário decodificar o caractere que se deseja apresentar, ou seja, passá-lo da representação binária convencional para outra que represente corretamente o caractere no *display*, de acordo com o seu arranjo de LEDs. Na tab. 5.2, é apresentado o valor binário para os números hexadecimais de 0 até F, considerando o segmento *a* como sendo o bit menos significativo (LSB). Caso o *display* esteja ligado a um PORT de 8 bits, para ligar o ponto (p) basta habilitar o 8º bit (MSB).

Tab. 5.2 - Valores para a decodificação de *displays* de 7 segmentos.

Dígito	Anodo comum		Catodo comum	
	gfedcba		gfedcba	
0	0b1000000	0x40	0b0111111	0x3F
1	0b1111001	0x79	0b0000110	0x06
2	0b0100100	0x24	0b1011011	0x5B
3	0b0110000	0x30	0b1001111	0x4F
4	0b0011001	0x19	0b1100110	0x66
5	0b0010010	0x12	0b1101101	0x6D
6	0b0000010	0x02	0b1111101	0x7D
7	0b1111000	0x78	0b0000111	0x07
8	0b0000000	0x00	0b1111111	0x7F
9	0b0011000	0x18	0b1100111	0x67
A	0b0001000	0x08	0b1110111	0x77
B	0b0000011	0x03	0b1111100	0x7C
C	0b1000110	0x46	0b0111001	0x39
D	0b0100001	0x21	0b1011110	0x5E
E	0b0000110	0x06	0b1111001	0x79
F	0b0001110	0x0E	0b1110001	0x71

Na fig. 5.13, é apresentado um fluxograma para mostrar um número hexadecimal (de 0 até F) em um *display* de 7 segmentos quando um botão é pressionado. Se o botão é mantido pressionado, o valor é constantemente alterado e, após chegar ao valor F, retorna a 0. Os programas em *assembly* e C são apresentados na sequência. Em relação ao fluxograma, a diferença entre eles é que o programa em *assembly* fez uso de uma sub-rotina para decodificar o número e mostrá-lo no *display*. O circuito microcontrolado empregado encontra-se na fig. 5.14.

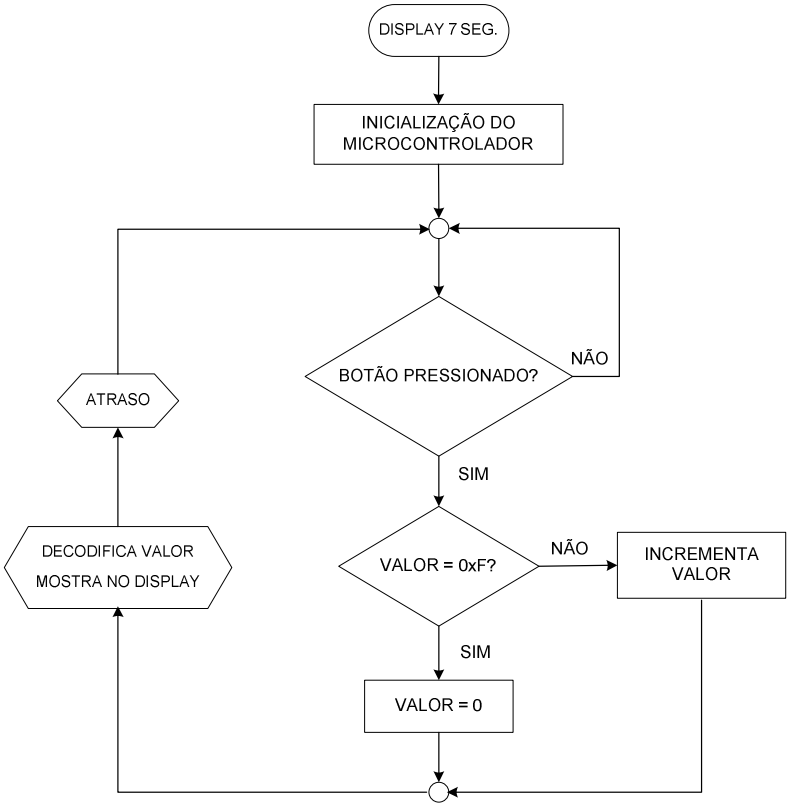
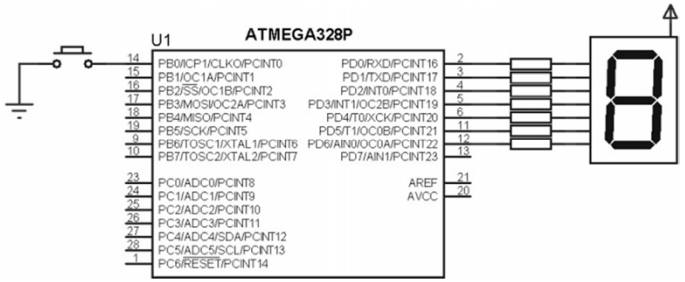


Fig. 5.13 – Fluxograma para apresentar um número hexadecimal de 0 até F quando um botão é pressionado.

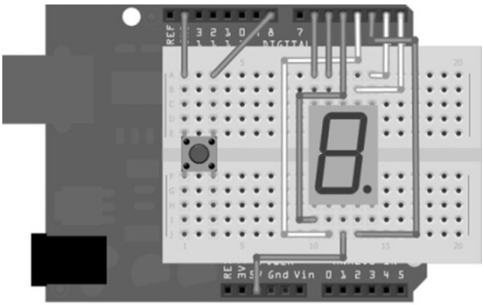
Display_7Seg.asm

```
//=============================================================================== //
//         ESCREVENDO EM UM DISPLAY DE 7 SEGMENTOS ANODO COMUM                    //
//   Toda vez que um botão é pressionado o valor do Display muda(0->F)            //
//   mantendo-se o botão pressionado o incremento é automático                    //
//=============================================================================== //
.equ BOTAO    = PB0   //BOTAO é o substituto de PB0 na programação
.equ DISPLAY  = PORTD //PORTD é onde está conectado o Display (seg a = LSB)
.def AUX      = R16;  //R16 tem agora o nome de AUX
//------------------------------------------------------------------------------
.ORG 0x000

Inicializacoes:
     LDI  AUX,0b11111110  //carrega AUX com o valor 0xFE (1 saída, 0 entrada)
     OUT  DDRB,AUX        //configura PORTB, PB0 entrada e PB1 .. PB7 saídas
     LDI  AUX,0xFF
     OUT  PORTB,AUX       //habilita o pull-up do PB0 (demais pinos em 1)
     OUT  DDRD, AUX       //PORTD como saída
     OUT  PORTD,AUX       //desliga o display
```

```
     /*Para utilizar os pinos PD0 e PD1 como I/O genérico no Arduino é necessário
                                        desabilitar as funções TXD e RXD desses pinos*/
     STS UCSR0B,R1   /*carrega o valor 0x00 (default de R1) em USCR0B,
                                                  como ele esta na SRAM, usa-se STS*/
//-------------------------------------------------------------------------------------
Principal:

     SBIC PINB,BOTAO      //verifica se o botão foi pressionado, senão
     RJMP Principal       //volta e fica preso no laço Principal

     CPI  AUX,0x0F        //compara se valor é máximo
     BRNE Incr            //se não for igual, incrementa; senão, zera valor
     LDI  AUX,0x00
     RJMP Decod

Incr:
     INC  AUX

Decod:
     RCALL Decodifica     //chama sub-rotina de decodificação
     RCALL Atraso         /*incremento automático do display se o botão ficar
                                                                        pressionado*/
     RJMP  Principal      //volta ler botão

//-------------------------------------------------------------------------------------
//SUB-ROTINA de atraso - Aprox. 0,2 s à 16 MHz
//-------------------------------------------------------------------------------------
Atraso:
     LDI   R19,16         //repete os laços abaixo 16 vezes
volta:
     DEC   R17            //decrementa R17
     BRNE  volta          //enquanto R17 > 0 fica decrementando R17
     DEC   R18            //decrementa R18
     BRNE  volta          //enquanto R18 > 0 volta a decrementar R17
     DEC   R19            //decrementa R19, começa com 0x02
     BRNE  volta
     RET
//-------------------------------------------------------------------------------------
//SUB-ROTINA que decodifica um valor de 0 -> 15 para o display
//-------------------------------------------------------------------------------------
Decodifica:

     LDI  ZH,HIGH(Tabela<<1)   /*carrega o endereço da tabela no registrador Z, de
                                              16 bits (trabalha como um ponteiro)*/
     LDI  ZL,LOW(Tabela<<1)    /*deslocando a esquerda todos os bits, pois o bit 0 é
                               para a seleção do byte alto ou baixo no end. de memória*/
     ADD  ZL,AUX               /*soma posição de memória correspondente ao nr. a
                                           apresentar na parte baixa do endereço*/
     BRCC le_tab               /*se houve Carry, incrementa parte alta do endereço,
                                                 senão lê diretamente a memória*/
     INC  ZH

le_tab:
     LPM  R0,Z                 //lê valor em R0
     OUT  DISPLAY,R0           //mostra no display
     RET
//-------------------------------------------------------------------------------------
//    Tabela p/ decodificar o display: como cada endereço da memória flash é de 16 bits,
//    acessa-se a parte baixa e alta na decodificação
//-------------------------------------------------------------------------------------
Tabela: .dw 0x7940, 0x3024, 0x1219, 0x7802, 0x1800, 0x0308, 0x2146, 0x0E06
//             1 0     3 2     5 4     7 6     9 8     B A     D C     F E
//=====================================================================================
```



a)



b)

Fig. 5.14 – Circuito para acionamento de um *display* de 7 segmentos anodo comum: a) esquemático e b) montagem no Arduino.

Display_7Seg.c

```
//================================================================================ //
//              ESCREVENDO EM UM DISPLAY DE 7 SEGMENTOS ANODO COMUM                 //
//================================================================================ //
#define F_CPU 16000000UL  //define a frequência do microcontrolador em 16MHz

#include <avr/io.h>       //definições do componente especificado
#include <util/delay.h>   //biblioteca para o uso das rotinas de _delay_
#include <avr/pgmspace.h> //biblioteca para poder gravar dados na memória flash

//Definições de macros - para o trabalho com os bits de uma variável
#define tst_bit(Y,bit_x) (Y&(1<<bit_x)) //testa o bit x da variável Y (retorna 0 ou 1)

#define DISPLAY  PORTD    //define um nome auxiliar para o display
#define BOTAO    PB0      //define PB0 com o nome de BOTAO

//variável gravada na memória flash
const unsigned char Tabela[] PROGMEM = {0x40, 0x79, 0x24, 0x30, 0x19, 0x12,
                          0x02, 0x78, 0x00, 0x18, 0x08, 0x03, 0x46, 0x21, 0x06, 0x0E};
//--------------------------------------------------------------------------------
```

```
int main()
{
     unsigned char valor = 0; //declara variável local

     DDRB = 0b11111110;   //PB0 como pino de entrada, os demais pinos como saída
     PORTB= 0x01;         //habilita o pull-up do PB0
     DDRD = 0xFF;         //PORTD como saída (display)
     PORTD= 0xFF;         //desliga o display
     UCSR0B = 0x00;       //PD0 e PD1 como I/O genérico, para uso no Arduino

     while(1)                        //laço infinito
     {
          if(!tst_bit(PINB,BOTAO))  //se o botão for pressionado executa
          {
               if(valor==0x0F)      //se o valor for igual a 0xF, zera o valor,
                    valor=0;
               else                 //se não o incrementa
                    valor++;

               //decodifica o valor e mostra no display, busca o valor na Tabela.
               DISPLAY = pgm_read_byte(&Tabela[valor]);

               _delay_ms(200); //atraso para incremento automático do nr. no display

          }//if botão

     }//laço infinito
}
//=============================================================================
```

O emprego de tabelas é muito usual e poderoso na programação de microcontroladores. Elas devem ser armazenadas na memória de programa para evitar o desperdício da memória RAM e da própria memória *flash*. A gravação de dados na memória RAM aumenta o tamanho do código porque a movimentação das constantes para as posições da RAM é realizada pelo programa. Isso significa que é duplamente prejudicial empregar a RAM para armazenar dados que não serão alterados durante o programa. Esse problema pode passar despercebido quando se programa em linguagem C. Com o compilador AVR-GCC, a gravação de dados na memória *flash* é feita com uso da biblioteca **pgmspace.h** e do comando **const ... PROGMEM** na declaração da variável, conforme apresentado no programa anterior.

É importante salientar que cada posição de memória *flash* do AVR ocupa 16 bits. Entretanto, o hardware permite o acesso a dados gravados por bytes individualmente. O bit 0 do registrador ZL informa se deve ser lido o byte baixo ou alto do endereço. Para isso, é preciso concatenar 2 bytes para cada posição de memória (ver a tabela do programa *assembly*

apresentado anteriormente). Em C, a programação é bem mais fácil e detalhes do microcontrolador não precisam ser conhecidos, como exigido ao se programar em *assembly*.

Exercícios:

5.13 – Elaborar um programa para apresentar em um *display* de 7 segmentos um número aleatório[25] entre 1 e 6 quando um botão for pressionado, ou seja, crie um dado eletrônico. Empregue o mesmo circuito da fig. 5.14.

5.14 – Elaborar um programa para apresentar nos LEDs da fig. 5.15 um número aleatório entre 1 e 6, formando os números de um dado (mesma lógica do exercício acima).

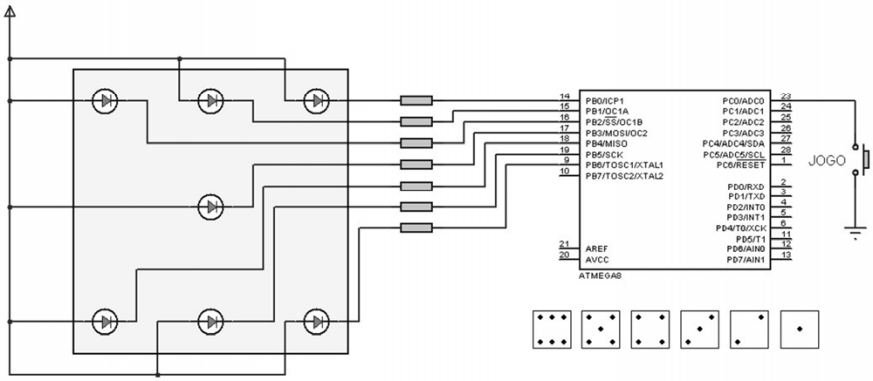


Fig. 5.15 – Dado eletrônico com LEDs.

Obs.: a pinagem do ATmega8 é igual a do ATmega328.

---

[25] Na verdade, criar um número puramente aleatório é difícil, o mais fácil é um pseudoaleatório. Neste exercício, o objetivo é não empregar as bibliotecas padrão do C. A ideia é utilizar o botão para gerar o evento de sorteio do número. Dessa forma, um contador pode ficar contando continuamente de 1 até 6 e, quando o botão for pressionado, um número da contagem será selecionado.

5.5 ACIONANDO LCDs 16 x 2

Os módulos LCDs são interfaces de saída muito úteis em sistemas microcontrolados. Estes módulos podem ser gráficos ou a caractere (alfanuméricos). Os LCDs comuns, tipo caractere, são especificados em número de linhas por colunas, sendo mais usuais as apresentações 16 × 2, 16 × 1, 20 × 2, 20 × 4, 8 × 2. Além disso, os módulos podem ser encontrados com *backlight* (LEDs para iluminação de fundo), facilitando a leitura em ambientes escuros. Os LCDs mais comuns empregam o CI controlador HD44780 da Hitachi com interface paralela. Há no mercado também LCDs com controle serial, sendo que novos LCDs são constantemente criados.

A seguir, será descrito o trabalho com o LCD de 16 caracteres por 2 linhas (ver o apêndice B); o uso de qualquer outro baseado no controlador HD44780 é similar. Na fig. 5.16, é apresentado o circuito microcontrolado com um *display* de 16 × 2.

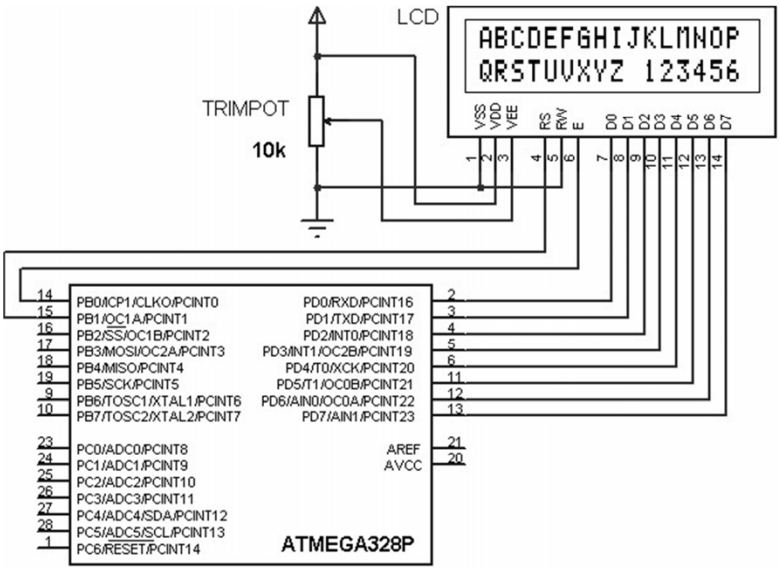


Fig. 5.16 – Circuito para acionamento de um LCD 16 × 2 usando 8 vias de dados.

Existem duas possibilidades de comunicação com o *display* da fig. 5.16. Uma é empregando 8 via de dados para a comunicação (D0-D7) e a outra, 4 vias de dados (D4-D7). Nesta última, o dado é enviado separadamente em duas partes (2 *nibbles*).

5.5.1 INTERFACE DE DADOS DE 8 BITS

Para ler ou escrever no *display* LCD com uma via de dados de 8 bits, é necessário a seguinte sequência de comandos:

1. Levar o pino R/W (*Read/Write*) para 0 lógico se a operação for de escrita e 1 lógico se for de leitura. Aterra-se esse pino se não há necessidade de monitorar a resposta do LCD (forma mais usual de trabalho).
2. Levar o pino RS (*Register Select*) para o nível lógico 0 ou 1 (instrução ou caractere).
3. Transferir os dados para a via de dados (8 bits).
4. Gerar um pulso de habilitação. Ou seja, levar o pino E (*Enable*) para 1 lógico e, após um pequeno tempo, para 0 lógico.
5. Empregar uma rotina de atraso entre as instruções ou fazer a leitura do *busy flag* (o bit 7 da linha de dados que indica que o *display* está ocupado) antes do envio da instrução, enviando-a somente quando esse *flag* for 0 lógico.

Os passos 1, 2 e 3 podem ser efetuados em qualquer sequência, pois o pulso de habilitação é que faz o controlador do LCD ler os dados dos seus pinos. É importante respeitar os tempos de resposta do LCD à transição dos sinais enviados ao mesmo[26].

Toda vez que a alimentação do LCD é ligada, deve ser executada uma rotina de inicialização. O LCD começa a responder aproximadamente 15 ms após a tensão de alimentação atingir 4,5 V. Como não se conhece o

[26] Para uma melhor compreensão sobre o assunto, consultar o manual do fabricante do LCD empregado.

tempo necessário para que ocorra a estabilização da tensão no circuito onde está colocado o LCD, pode ser necessário frações bem maiores de tempo para que o LCD possa responder a comandos. Muitas vezes, esse detalhe é esquecido e o LCD não funciona adequadamente. Para corrigir esse problema, basta colocar uma rotina de atraso suficientemente grande na inicialização do LCD. Na fig. 5.17, é apresentado o fluxograma de inicialização do LCD conforme especificação da Hitachi. Se desejado, o *busy flag* pode ser lido após o ajuste do modo de utilização do *display*. Os comandos para o LCD são detalhado no apêndice B.

A seguir são apresentadas as rotinas de escrita no LCD em conjunto com um programa exemplo que escreve na linha superior do LCD "ABCDEFGHIJKLMNOP" e "QRSTUVXYZ 123456" na linha inferior (circuito da fig. 5.16). Muitos programadores não utilizam a rotina de inicialização completa do diagrama da fig. 5.17, respeitando apenas o tempo de resposta do *display*, seguido do modo de utilização e dos outros controles, prática que geralmente funciona.

Para a visualização dos caracteres é imprescindível o emprego do *trimpot* (fig. 5.16) para ajuste do contraste do *display* de cristal líquido.

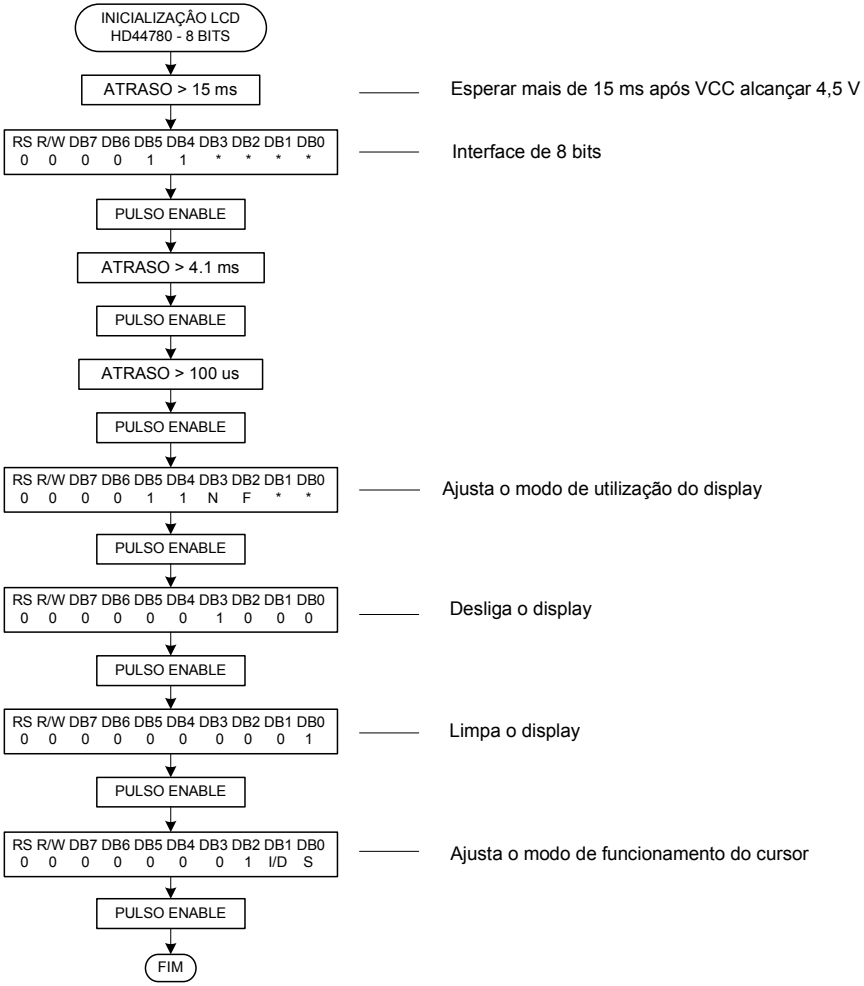


Fig. 5.17 – Rotina de inicialização de 8 bits para um LCD com base no CI HD44780.

LCD_8bits.c

```
//=============================================================================== //
//      ACIONANDO UM DISPLAY DE CRISTAL LIQUIDO DE 16x2                          //
//                                                                               //
//              Interface de dados de 8 bits                                     //
//=============================================================================== //
#define F_CPU 16000000UL  //define a frequência do microcontrolador - 16MHz

#include <avr/io.h>       //definições do componente especificado
#include <util/delay.h>   //biblioteca para o uso das rotinas de delay
#include <avr/pgmspace.h> //uso de funções para salvar dados na memória de programa

//Definições de macros - empregadas para o trabalho com o bits
#define   set_bit(Y,bit_x) (Y|=(1<<bit_x))    //ativa o bit x da variável Y
#define   clr_bit(Y,bit_x) (Y&=~(1<<bit_x))   //limpa o bit x da variável Y
#define   tst_bit(Y,bit_x) (Y&(1<<bit_x))     //testa o bit x da variável Y
#define   cpl_bit(Y,bit_x) (Y^=(1<<bit_x))    //troca o estado do bit x da variável Y

//para uso no LCD (deve estar na mesma linha)
#define pulso_enable() _delay_us(1); set_bit(CONTR_LCD,E); _delay_us(1);
                                                clr_bit(CONTR_LCD,E); _delay_us(45)
#define DADOS_LCD    PORTD     //8 bits de dados do LCD na porta D
#define CONTR_LCD    PORTB     //os pinos de controle estão no PORTB
#define RS           PB1       //pino de instrução ou dado para o LCD
#define E            PB0       //pino de enable do LCD

//mensagem armazenada na memória flash
const unsigned char msg1[] PROGMEM = "ABCDEFGHIJKLMNOP";

//---------------------------------------------------------------------------------
//Sub-rotina para enviar caracteres e comandos ao LCD
//---------------------------------------------------------------------------------
void cmd_LCD(unsigned char c, char cd)//c é o dado  e cd indica se é instrução ou caractere
{
     DADOS_LCD = c;

     if(cd==0)
          clr_bit(CONTR_LCD,RS);    //RS = 0
     else
          set_bit(CONTR_LCD,RS);    //RS = 1

     pulso_enable();

     //se for instrução de limpeza ou retorno de cursor espera o tempo necessário
     if((cd==0) && (c<4))
          _delay_ms(2);
}
//---------------------------------------------------------------------------------
//Sub-rotina de inicialização do LCD
//---------------------------------------------------------------------------------
void inic_LCD_8bits()//sequência ditada pelo fabricando do circuito de controle do LCD
{
     clr_bit(CONTR_LCD,RS);//o LCD será só escrito então R/W é sempre zero

     _delay_ms(15);  /*tempo para estabilizar a tensão do LCD, após VCC ultrapassar
                                              4.5 V (pode ser bem maior na prática)*/

     DADOS_LCD = 0x38;    //interface 8 bits, 2 linhas, matriz 7x5 pontos

     pulso_enable();      //enable respeitando os tempos de resposta do LCD
     _delay_ms(5);
     pulso_enable();
     _delay_us(200);
```

```
     pulso_enable();
     pulso_enable();

     cmd_LCD(0x08,0);     //desliga LCD
     cmd_LCD(0x01,0);     //limpa todo o display
     cmd_LCD(0x0C,0);     //mensagem aparente cursor inativo não piscando
     cmd_LCD(0x80,0);     //escreve na primeira posição a esquerda - 1ª linha
}
//---------------------------------------------------------------------------------
//Sub-rotina de escrita no LCD
//---------------------------------------------------------------------------------
void escreve_LCD(char *c)
{
   for (; *c!=0;c++) cmd_LCD(*c,1);
}
//---------------------------------------------------------------------------------
int main()
{
    unsigned char i;

     DDRB = 0xFF;    //PORTB como saída
     DDRD = 0xFF;    //PORTD como saída
     UCSR0B = 0x00;  //habilita os pinos PD0 e PD1 como I/O para uso no Arduino

     inic_LCD_8bits();    //inicializa o LCD

     for(i=0;i<16;i++)    //enviando caractere por caractere
          cmd_LCD(pgm_read_byte(&msg1[i]),1); //lê na memória flash e usa cmd_LCD

     cmd_LCD(0xC0,0);     //desloca o cursor para a segunda linha do LCD
     escreve_LCD("QRSTUVXYZ 123456");//a cadeia de caracteres é criada na RAM

     for(;;);             //laço infinito
}
//=================================================================================
```

5.5.2 INTERFACE DE DADOS DE 4 BITS

Nesta seção, o trabalho com o LCD e suas rotinas serão melhores detalhados, visto que utilizar uma interface de dados de 8 bits para um LCD de 16×2 não é recomendado. Isso se deve ao uso excessivo de vias para o acionamento do *display* (10 ou 11). O emprego de 4 bits de dados libera 4 pinos de I/O do microcontrolador para outras atividades, além de diminuir o número de trilhas necessárias na placa de circuito impresso. O custo é um pequeno aumento na complexidade do programa de controle do LCD, o que consome alguns bytes a mais de programação.

Para ler ou escrever no *display* LCD com uma via de dados de 4 bits, é necessário a seguinte sequência de comandos:

1. Levar o pino R/W (*Read/Write*) para 0 lógico se a operação for de escrita e 1 lógico se for de leitura. Aterra-se esse pino se não há necessidade de monitorar a resposta do LCD (forma mais usual de trabalho).
2. Levar o pino RS (*Register Select*) para o nível lógico 0 ou 1 (instrução ou caractere).
3. Transferir a parte mais significativa dos dados para a via de dados (4 bits mais significativos (MSB) – *nibble* maior).
4. Gerar um pulso de habilitação. Ou seja, levar o pino E (*Enable*) para 1 lógico e após um pequeno tempo de espera para 0 lógico.
5. Transferir a parte menos significativa dos dados para a via de dados (4 bits menos significativos (LSB) – *nibble* menor).
6. Gerar outro pulso de habilitação.
7. Empregar uma rotina de atraso entre as instruções ou fazer a leitura do *busy flag* (o bit 7 da linha de dados que indica que o *display* está ocupado) antes do envio da instrução, enviando-a somente quando esse *flag* for 0 lógico.

Os passos 1, 2 e 3 podem ser efetuados em qualquer sequência, pois o pulso de habilitação é que faz o controlador do LCD ler os dados dos seus pinos.

Na fig. 5.18, é apresentado o circuito microcontrolado com o LCD 16×2 com via de dados de 4 bits (conexão igual a do módulo LCD *shield* da Ekitszone[27]). Na sequência, mostram-se o fluxograma de inicialização do *display*, de acordo com a Hitachi, e o programa de controle do LCD; os arquivos com os programas para o trabalho com o LCD foram organizados de forma estruturada (ver a seção 4.5.11). O resultado prático é visto na fig. 5.20.

[27] www.ekitszone.com. Existem outros módulos LCD disponíveis no mercado, depende somente da escolha do usuário. Se desejado o circuito pode ser montado em uma matriz de contatos.

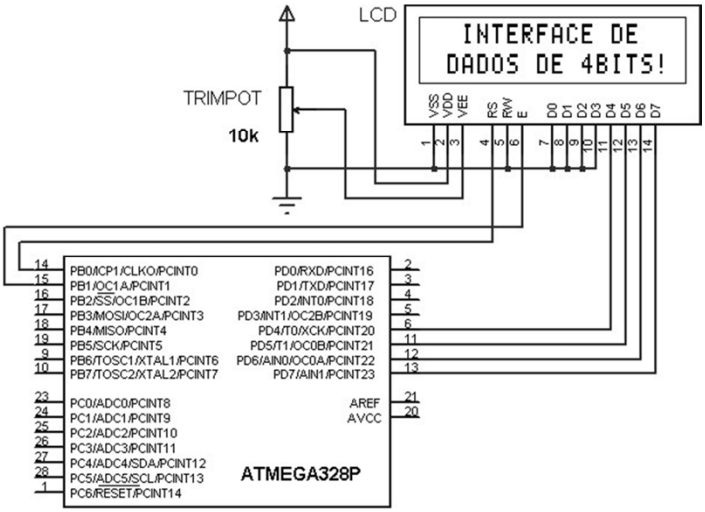


Fig. 5.18 – Circuito para acionamento de um LCD 16 × 2 com interface de dados de 4 bits.

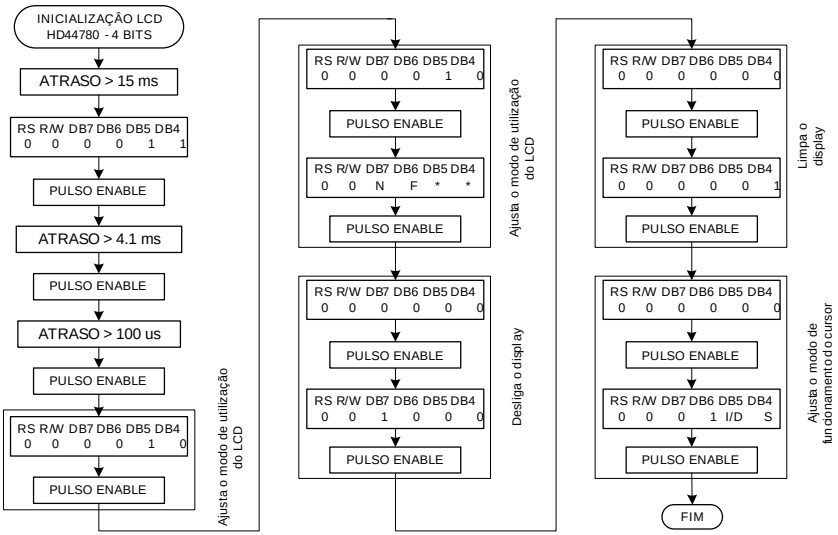


Fig. 5.19 – Rotina de inicialização de 4 bits para um LCD com base no CI HD44780.

def_principais.h (arquivo de cabeçalho do programa principal)

```
#ifndef _DEF_PRINCIPAIS_H
#define _DEF_PRINCIPAIS_H

#define F_CPU 16000000UL  //define a frequência do microcontrolador - 16MHz

#include <avr/io.h>     //definições do componente especificado
#include <util/delay.h> //biblioteca para o uso das rotinas de _delay_ms e _delay_us()
#include <avr/pgmspace.h>//para a gravação de dados na memória flash

//Definições de macros para o trabalho com bits
#define set_bit(y,bit) (y|=(1<<bit))  //coloca em 1 o bit x da variável Y
#define clr_bit(y,bit) (y&=~(1<<bit)) //coloca em 0 o bit x da variável Y
#define cpl_bit(y,bit) (y^=(1<<bit))  //troca o estado lógico do bit x da variável Y
#define tst_bit(y,bit) (y&(1<<bit))   //retorna 0 ou 1 conforme leitura do bit

#endif
```

LCD_4bits.c (programa principal)

```
//=============================================================================== //
//         ACIONANDO UM DISPLAY DE CRISTAL LIQUIDO DE 16x2                       //
//                   Interface de dados de 4 bits                                //
//=============================================================================== //

#include "def_principais.h"    //inclusão do arquivo com as principais definições
#include "LCD.h"

//definição para acessar a memória flash
prog_char mensagem[] = " DADOS DE 4BITS!\0"; //mensagem armazenada na memória flash

//------------------------------------------------------------------------------
int main()
{
     DDRD = 0xFF;                   //PORTD como saída
     DDRB = 0xFF;                   //PORTB como saída

     inic_LCD_4bits();              //inicializa o LCD
     escreve_LCD("  INTERFACE DE"); //string armazenada na RAM
     cmd_LCD(0xC0,0);               //desloca cursor para a segunda linha
     escreve_LCD_Flash(mensagem);   //string armazenada na flash

     for(;;){}                      //laço infinito, aqui vai o código principal
}
//==============================================================================
```

LCD.h (arquivo de cabeçalho do LCD.c)

```
#ifndef _LCD_H
#define _LCD_H

#include "def_principais.h"

#define DADOS_LCD    PORTD//4 bits de dados do LCD no PORTD
#define nibble_dados  1   /*0 para via de dados do LCD nos 4 LSBs do PORT
                          empregado (Px0-D4, Px1-D5, Px2-D6, Px3-D7), 1 para via de
                          dados do LCD nos 4 MSBs do PORT empregado (Px4-D4, Px5-D5,
                          Px6-D6, Px7-D7) */
#define CONTR_LCD    PORTB//PORT com os pinos de controle do LCD (pino R/W em 0).
#define E            PB1  //pino de habilitação do LCD (enable)
#define RS           PB0  //pino para informar se o dado é uma instrução ou caractere

#define tam_vetor    5    //número de digitos individuais para a conversão por ident_num()
#define conv_ascii   48   /*48 se ident_num() deve retornar um número no formato ASCII (0 para
                                                                        formato normal)*/

//sinal de habilitação para o LCD
#define pulso_enable() _delay_us(1); set_bit(CONTR_LCD,E); _delay_us(1);
                                                    clr_bit(CONTR_LCD,E); _delay_us(45)
//protótipo das funções
void cmd_LCD(unsigned char c, char cd);
void inic_LCD_4bits();
void escreve_LCD(char *c);
void escreve_LCD_Flash(const char *c);

void ident_num(unsigned int valor, unsigned char *disp);

#endif
```

LCD.c (funções para o LCD)

```
//============================================================================== //
//   Sub-rotinas para o trabalho com um LCD 16x2 com via de dados de 4 bits       //
//        Controlador HD44780  - Pino R/W aterrado                                //
//        A via de dados do LCD deve ser ligado aos 4 bits mais significativos ou //
//        aos 4 bits menos significativos do PORT do uC                           //
//============================================================================== //

#include "LCD.h"

//---------------------------------------------------------------------------------
// Sub-rotina para enviar caracteres e comandos ao LCD com via de dados de 4 bits
//---------------------------------------------------------------------------------
//c é o dado  e cd indica se é instrução ou caractere (0 ou 1)
void cmd_LCD(unsigned char c, char cd)
{
     if(cd==0)                      //instrução
          clr_bit(CONTR_LCD,RS);
     else                           //caractere
          set_bit(CONTR_LCD,RS);

                     //primeiro nibble de dados - 4 MSB
     #if (nibble_dados)//compila o código para os pinos de dados do LCD nos 4 MSB do PORT
          DADOS_LCD = (DADOS_LCD & 0x0F)|(0xF0 & c);
     #else             //compila o código para os pinos de dados do LCD nos 4 LSB do PORT
          DADOS_LCD = (DADOS_LCD & 0xF0)|(c>>4);
     #endif

     pulso_enable();
```

```
     //segundo nibble de dados - 4 LSB
     #if (nibble_dados)   //compila o código para os pinos de dados do LCD nos 4 MSB do PORT
          DADOS_LCD = (DADOS_LCD & 0x0F) | (0xF0 & (c<<4));
     #else                //compila o código para os pinos de dados do LCD nos 4 LSB do PORT
          DADOS_LCD = (DADOS_LCD & 0xF0) | (0x0F & c);
     #endif

     pulso_enable();

     if((cd==0) && (c<4)) //se for instrução de retorno ou limpeza espera LCD estar pronto
          _delay_ms(2);
}
//---------------------------------------------------------------------------
//Sub-rotina para inicialização do LCD com via de dados de 4 bits
//---------------------------------------------------------------------------
void inic_LCD_4bits()//sequência ditada pelo fabricando do circuito integrado HD44780
{                    //o LCD será só escrito. Então, R/W é sempre zero.

     clr_bit(CONTR_LCD,RS);//RS em zero indicando que o dado para o LCD será uma instrução
     clr_bit(CONTR_LCD,E);//pino de habilitação em zero

     _delay_ms(20);       /*tempo para estabilizar a tensão do LCD, após VCC
                                 ultrapassar 4.5 V (na prática pode ser maior).*/
     //interface de 8 bits
     #if (nibble_dados)
          DADOS_LCD = (DADOS_LCD & 0x0F) | 0x30;
     #else
          DADOS_LCD = (DADOS_LCD & 0xF0) | 0x03;
     #endif

     pulso_enable();           //habilitação respeitando os tempos de resposta do LCD
     _delay_ms(5);
     pulso_enable();
     _delay_us(200);
     pulso_enable();           //até aqui ainda é uma interface de 8 bits.

     //interface de 4 bits, deve ser enviado duas vezes (a outra está abaixo)
     #if (nibble_dados)
          DADOS_LCD = (DADOS_LCD & 0x0F) | 0x20;
     #else
          DADOS_LCD = (DADOS_LCD & 0xF0) | 0x02;
     #endif

     pulso_enable();
     cmd_LCD(0x28,0); //interface de 4 bits 2 linhas (aqui se habilita as 2 linhas)
                      //são enviados os 2 nibbles (0x2 e 0x8)
     cmd_LCD(0x08,0); //desliga o display
     cmd_LCD(0x01,0); //limpa todo o display
     cmd_LCD(0x0C,0); //mensagem aparente cursor inativo não piscando
     cmd_LCD(0x80,0); //inicializa cursor na primeira posição a esquerda - 1a linha
}
//---------------------------------------------------------------------------
//Sub-rotina de escrita no LCD -  dados armazenados na RAM
//---------------------------------------------------------------------------
void escreve_LCD(char *c)
{
   for (; *c!=0;c++) cmd_LCD(*c,1);
}
//---------------------------------------------------------------------------
//Sub-rotina de escrita no LCD - dados armazenados na FLASH
//---------------------------------------------------------------------------
void escreve_LCD_Flash(const char *c)
{
   for (;pgm_read_byte(&(*c))!=0;c++) cmd_LCD(pgm_read_byte(&(*c)),1);
}
```

```
//-------------------------------------------------------------------------------------
//Conversão de um número em seus digitos individuais – função auxiliar
//-------------------------------------------------------------------------------------
void ident_num(unsigned int valor, unsigned char *disp)
{
     unsigned char n;

     for(n=0; n<tam_vetor; n++)
          disp[n] = 0 + conv_ascii;      //limpa vetor para armazenagem dos digitos

     do
     {
       *disp = (valor%10) + conv_ascii;  //pega o resto da divisão por 10
        valor /=10;                      //pega o inteiro da divisão por 10
        disp++;

     }while (valor!=0);
}
//-------------------------------------------------------------------------------------
```

É fundamental compreender as funções apresentadas para a programação do LCD. As funções foram desenvolvidas para serem flexíveis quanto à conexão dos pinos do LCD ao microcontrolador, a única ressalva é quanto a disposição dos 4 pinos de dados do LCD (D4-D7), os quais devem ser conectados em um *nibble* alto ou em um *nibble* baixo do PORT utilizado. As definições dos pinos é feita no arquivo LCD.h.

Fig. 5.20 – Resultado do programa para controle de um LCD 16×2 com interface de dados de 4 bits (módulo LCD – *shield*, da Ekitszone).

A principal função para o controle do LCD é a **cmd_LCD(dado, 0 ou 1)**, que recebe dois parâmetros: o dado que se deseja enviar ao LCD e o

número 0 ou 1; onde 0 indica que o dado é uma instrução e 1 indica que o dado é um caractere.

A função **inic_LCD_4bits( )** deve ser utilizada no início do programa principal para a correta inicialização do LCD. Existe uma sequência de comandos que deve ser seguida para que o LCD possa funcionar corretamente.

A função **escreve_LCD("frase")** recebe uma *string*, ou seja um conjunto de caracteres. Como na programação em C toda *string* é finalizada com o caractere nulo (0), essa função se vale desse artifício para verificar o final da *string*. Deve-se ter cuidado ao se utilizar essa função, porque a *string* é armazenada na memória RAM do microcontrolador, o que pode limitar a memória disponível para o programa. Para evitar esse problema, pode-se utilizar a função **escreve_LCD_Flash(frase)**, onde a frase, previamente declarada no programa, é armazenada na memória *flash*.

Uma vez inicializado o LCD, se escolhe em qual posição dele se deseja escrever. Cada vez que um caractere é escrito, o cursor é automaticamente deslocado para a próxima posição de escrita, à direita ou à esquerda conforme inicialização (ver a tabela B.3 do apêndice B). Assim, é importante entender como mudar a posição do cursor antes de escrever o caractere. Na fig. 5.21, são apresentados os endereços correspondentes a cada caractere do LCD 16 × 2.

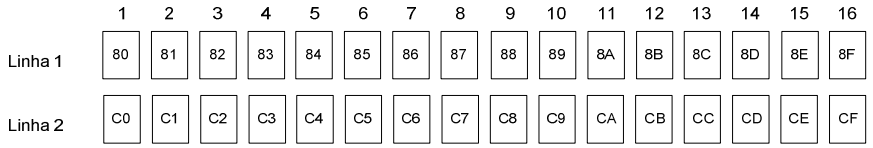


Fig. 5.21 – Endereços para escrita num LCD 16 × 2.

Por exemplo, quando se deseja escrever o caractere A na 6ª posição da linha superior (linha 1), deve ser empregado o seguinte código:

```
cmd_LCD(0x85,0);     //desloca cursor para o endereço 0x86
cmd_LCD('A',1);      //escrita do caractere
```

Para escrever o conjunto de caracteres “Alo mundo!” na linha inferior começando na terceira posição, deve-se empregar o código:

```
cmd_LCD(0xC2,0);              //desloca cursor para o endereço 0xC2
escreve_LCD(“Alo mundo!”);    //escrita da string
```

A mensagem não aparecerá caso se escreva em uma posição que não exista na tela do LCD. Se outro LCD for empregado, como por exemplo um 20 × 4 (20 caracteres por 4 linhas) o endereçamento será diferente. Na fig. 5.22, são apresentados os endereços dos caracteres para um LCD 20 × 4. A linha 3 é continuação da linha 1 e a linha 4, continuação da linha 2. Na dúvida, o manual do fabricante deve ser consultado.

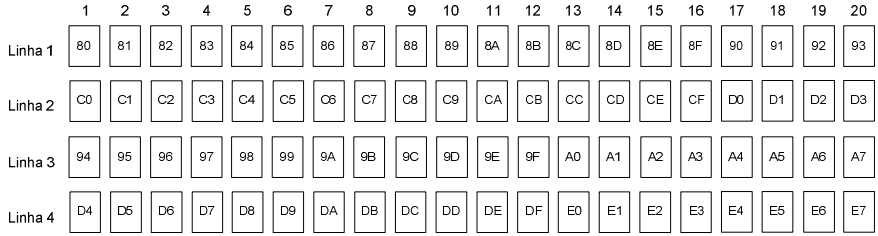


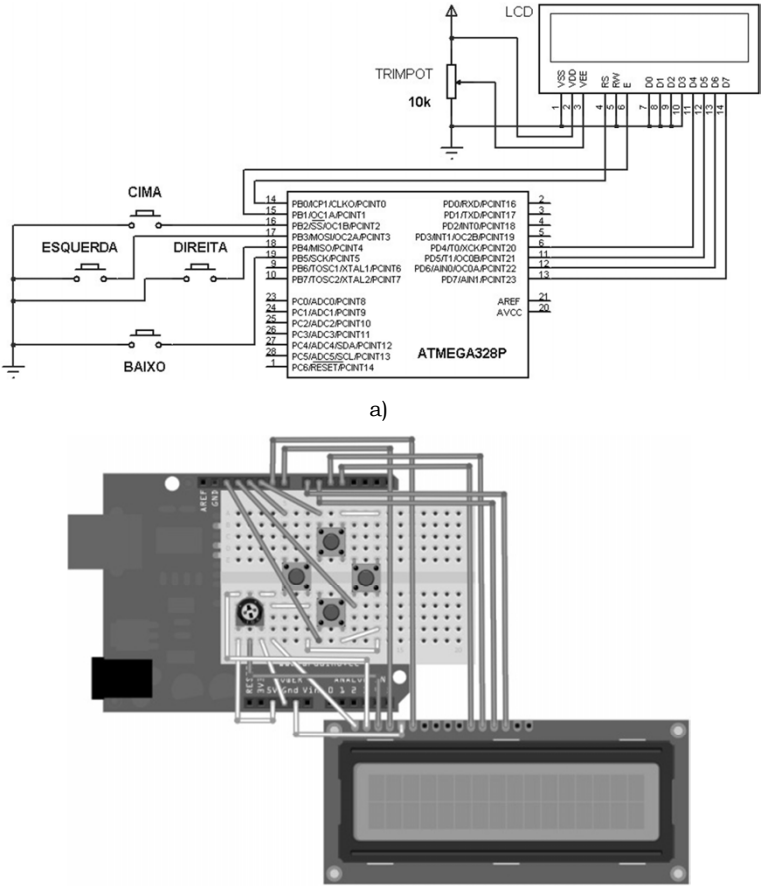
Fig. 5.22 – Endereços para a escrita num LCD 20 × 4.

Exercícios:

5.15 – Elaborar um programa para deslocar um caractere ‘ * ’ (asterisco) no LCD da fig. 5.18, da esquerda para a direita, ao chegar ao final da linha o caractere deve retornar (vai e vem).

5.16 – Repetir o exercício 5.15 empregando as duas linhas do LCD. Ao chegar ao final da linha superior, o asterisco começa na linha inferior (endereço 0xD3). Dessa forma, na linha superior o asterisco se desloca da esquerda para a direita e na linha inferior, da direita para a esquerda.

5.17 – Elaborar um programa para realizar o movimento de um cursor num LCD 16 × 2 com o uso de 4 botões, conforme fig. 5.23.



a)

b)

Fig. 5.23 – Exercício 5.17: a) esquemático e b) montagem no Arduino.

5.5.3 CRIANDO NOVOS CARACTERES

Os códigos dos caracteres recebidos pelo LCD são armazenados em uma RAM chamada DDRAM (*Data Display* RAM), transformados em um caractere no padrão matriz de pontos e apresentados na tela do LCD (ver a tabela no apêndice B para o conjunto de caracteres do HD44780). Para produzir os caracteres nesse padrão, o módulo LCD incorpora uma CGROM (*Character Generator* ROM). Os LCDs também possuem uma

CGRAM (*Character Generator* RAM) para a gravação de caracteres especiais. Oito caracteres programáveis podem ser escritos na CGRAM e apresentam os códigos fixos 0x00 a 0x07. Os dados da CGRAM são apresentados em um mapa de bits de 8 bytes, dos quais se utilizam 7, com 5 bits cada, sendo 1 byte reservado para o cursor. Dessa forma, qualquer caractere é representado por uma matriz de pontos 7 × 5. Na fig. 5.24, apresenta-se o mapa de bits para o símbolo Δ.

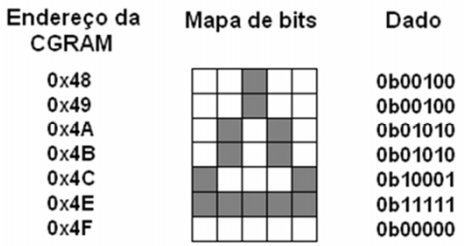


Fig. 5.24 – Gravação do símbolo Δ na CGRAM, matriz 5 × 7. Esse caractere será selecionado pelo código 0x01.

Como há espaço para 8 caracteres, estão disponíveis 64 bytes na CGRAM. Na tab. 5.3, são apresentados os endereços base onde devem ser criados os novos caracteres e os seus respectivos códigos para escrita no *display*. Na fig. 5.24, por exemplo, o símbolo Δ será selecionado pelo código 0x01, pois o mesmo possui o endereço base 0x48 da CGRAM.

Tab. 5.3 – Endereço para criação de caracteres novos e seus códigos de chamada.

Endereço base	0x40	0x48	0x50	0x58	0x60	0x68	0x70	0x78
Código do caractere	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07

A seguir, apresenta-se o programa que cria dois caracteres novos, um no endereço 0x40 e o outro no 0x48. Após criados, os mesmos possuem o códigos 0x00 e 0x01, respectivamente (ver a tabela 5.3). Na fig. 5.25, apresenta-se o resultado para o LCD 16 × 2.

LCD_4bits_2new_caract.c

```
//=============================================================================== //
//                   CRIANDO CARACTERES PARA O LCD 16x2                          //
//                        Via de dados de 4 bits                                 //
//=============================================================================== //

#include "def_principais.h"//inclusão do arquivo com as principais definições
#include "LCD.h"

//informação para criar caracteres novos armazenada na memória flash
const unsigned char carac1[] PROGMEM = {0b01110,//Ç
                                        0b10001,
                                        0b10000,
                                        0b10000,
                                        0b10101,
                                        0b01110,
                                        0b10000};

const unsigned char carac2[] PROGMEM = {0b00100,//Delta
                                        0b00100,
                                        0b01010,
                                        0b01010,
                                        0b10001,
                                        0b11111,
                                        0b00000};
//-------------------------------------------------------------------------------
int main()
{
     unsigned char k;

     DDRD = 0xFF;         //PORTD como saída
     DDRB = 0xFF;         //PORTB como saída

     inic_LCD_4bits();    //inicializa o LCD

     cmd_LCD(0x40,0);     //endereço base para gravar novo segmento
     for(k=0;k<7;k++)     //grava 8 bytes na DDRAM começando no end. 0x40
          cmd_LCD(pgm_read_byte(&carac1[k]),1);
     cmd_LCD(0x00,1);     /*apaga última posição do end. da CGRAM para evitar algum
                                                                    dado espúrio*/

     cmd_LCD(0x48,0);     //endereço base para gravar novo segmento
     for(k=0;k<7;k++)     //grava 8 bytes na DDRAM começando no end. 0x48
          cmd_LCD(pgm_read_byte(&carac2[k]),1);
     cmd_LCD(0x00,1);     /*apaga última posição do end. da CGRAM para evitar algum
                                                                    dado espúrio*/
     cmd_LCD(0x80,0);     //endereça a posição para escrita dos caracteres
     cmd_LCD(0x00,1);     //apresenta primeiro caractere 0x00
     cmd_LCD(0x01,1);     //apresenta segundo  caractere 0x01

     for(;;);             //laço infinito
}
//===============================================================================
```

Para usar a diretiva PROGMEM e poder gravar dados na memória *flash*, é necessário incluir a biblioteca **pgmspace.h** no arquivo **def_principais.h** apresentado anteriormente: `#include <avr/pgmspace.h>`

Fig. 5.25 - Dois caracteres novos: Ç e Δ.

O código citado anteriormente apresenta uma forma de criar caracteres individualmente. Em uma programação mais eficiente, uma única matriz de informação deve ser utilizada.

5.5.4 NÚMEROS GRANDES

Utilizar 4 caracteres do LCD 16×2 para criar números grandes é uma técnica interessante de escrita. Ela pode ser muito útil, por exemplo, para a representação de horas. Com um pouco de raciocínio, utilizando os caracteres disponíveis (tab. B.4 do apêndice B) e criando os 8 permitidos, é possível montar os dígitos de 0 até 9 empregando 4 caracteres para cada número.

Na fig. 5.26, são apresentados os 8 caracteres a serem criados e seus respectivos códigos, os quais, quando organizados adequadamente em conjunto com os caracteres existentes para o LCD, permitem formar os números grandes, conforme fig. 5.27.

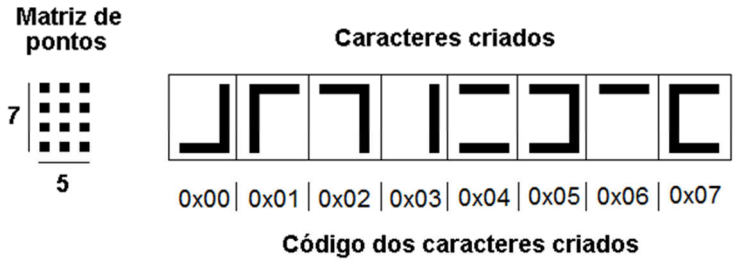


Fig. 5.26 – Caracteres criados para se poder escrever números grandes num LCD 16×2.

Fig. 5.27 – Organização de 4 caracteres para formar números grandes em um LCD 16×2.

Para mostrar um número no *display*, é necessário escrever dois caracteres na linha superior do LCD e os outros dois na linha inferior, de acordo com os caracteres que formam cada número. A seguir, é apresentada uma matriz com os códigos para a formação dos números grandes conforme os códigos dos caracteres da fig. 5.25 e os disponíveis na tab. B.4 do apêndice B. Cada linha da matriz representa os 4 caracteres necessários para gerar o número.

```
//===================================================================
unsigned char Nr_Grande[10][4] = {{0x01, 0x02, 0x4C, 0x00}, //nr. 0
                                  {0x20, 0x7C, 0x20, 0x7C}, //nr. 1
                                  {0x04, 0x05, 0x4C, 0x5F}, //nr. 2
                                  {0x06, 0x05, 0x5F, 0x00}, //nr. 3
                                  {0x4C, 0x00, 0x20, 0x03}, //nr. 4
                                  {0x07, 0x04, 0x5F, 0x00}, //nr. 5
                                  {0x07, 0x04, 0x4C, 0x00}, //nr. 5
                                  {0x06, 0x02, 0x20, 0x03}, //nr. 7
                                  {0x07, 0x05, 0x4C, 0x00}, //nr. 8
                                  {0x07, 0x05, 0x20, 0x03}};//nr. 9
//===================================================================
```

Cada número é escrito da seguinte forma: desloca-se o cursor de escrita para o endereço do *display* correspondente a primeira linha; escreve-se dois caracteres; desloca-se novamente o cursor, desta vez para a segunda linha, embaixo do primeiro caractere escrito; então, escreve-se os últimos dois caracteres. Na fig. 5.28, é apresentado o exemplo de um fluxograma simplificado para a escrita do número 5 com começo no endereço de caractere 0x80 (lado superior esquerdo do LCD, ver a fig. 5.21), considerando-se que os caracteres estão organizados conforme a tabela acima. Na sequência, apresenta-se o programa.

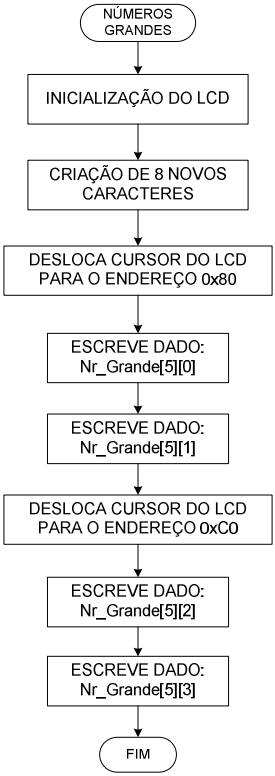


Fig. 5.28 – Fluxograma simplificado para apresentar o número grande 5 em um LCD 16 × 2.

LCD_4bits_big_number.c

```
//=============================================================================== //
//                  CRIANDO NÚMEROS GRANDES PARA O LCD 16x2                      //
//=============================================================================== //
#include "def_principais.h"    //inclusão do arquivo com as principais definições
#include "LCD.h"
//informações para criar novos caracteres, armazenadas na memória flash
const unsigned char novos_caract[] PROGMEM={0b00000001, 0b00000001, 0b00000001,
                        0b00000001, 0b00000001, 0b00000001, 0b00011111, //0
                        0b00011111, 0b00010000, 0b00010000, 0b00010000,
                        0b00010000, 0b00010000, 0b00010000, //1
                        0b00011111, 0b00000001, 0b00000001, 0b00000001,
                        0b00000001, 0b00000001, 0b00000001, //2
                        0b00000001, 0b00000001, 0b00000001, 0b00000001,
                        0b00000001, 0b00000001, 0b00000001, //3
                        0b00011111, 0b00000000, 0b00000000, 0b00000000,
                        0b00000000, 0b00000000, 0b00011111, //4
                        0b00011111, 0b00000001, 0b00000001, 0b00000001,
                        0b00000001, 0b00000001, 0b00011111, //5
```

```
                        0b00011111, 0b00000000, 0b00000000, 0b00000000,
                        0b00000000, 0b00000000, 0b00000000, //6
                        0b00011111, 0b00010000, 0b00010000, 0b00010000,
                        0b00010000, 0b00010000, 0b00011111};//7

const unsigned char nr_grande[10][4] PROGMEM= {{0x01, 0x02, 0x4C, 0x00}, //nr. 0
                                               {0x20, 0x7C, 0x20, 0x7C}, //nr. 1
                                               {0x04, 0x05, 0x4C, 0x5F}, //nr. 2
                                               {0x06, 0x05, 0x5F, 0x00}, //nr. 3
                                               {0x4C, 0x00, 0x20, 0x03}, //nr. 4
                                               {0x07, 0x04, 0x5F, 0x00}, //nr. 5
                                               {0x07, 0x04, 0x4C, 0x00}, //nr. 5
                                               {0x06, 0x02, 0x20, 0x03}, //nr. 7
                                               {0x07, 0x05, 0x4C, 0x00}, //nr. 8
                                               {0x07, 0x05, 0x20, 0x03}};//nr. 9
//-------------------------------------------------------------------------------------
void cria_novos_caract()//criação dos 8 novos caracteres
{
     unsigned char i, k, j=0, n=0x40;

     for(i=0;i<8;i++)
     {
          cmd_LCD(n,0);             //endereço base para gravar novo segmento
          for(k=0;k<7;k++)
               cmd_LCD(pgm_read_byte(&novos_caract[k+j]),1);
          cmd_LCD(0x00,1);/*apaga ultima posição do end. da CGRAM para evitar algum
                                                                        dado espúrio*/
          j += 7;
          n += 8;
     }
}
//-------------------------------------------------------------------------------------
void escreve_BIG(unsigned char end, unsigned char nr) //escreve um número grandes no LCD
{
     cmd_LCD(end,0);               //endereço de início de escrita (1a linha)
     cmd_LCD(pgm_read_byte(&nr_grande[nr][0]),1);
     cmd_LCD(pgm_read_byte(&nr_grande[nr][1]),1);
     cmd_LCD(end+64,0);            //desloca para a 2a linha
     cmd_LCD(pgm_read_byte(&nr_grande[nr][2]),1);
     cmd_LCD(pgm_read_byte(&nr_grande[nr][3]),1);
}
//-------------------------------------------------------------------------------------
int main()
{
     DDRD = 0xFF;                       //PORTD como saída
     DDRB = 0xFF;                       //PORTB como saída
     inic_LCD_4bits();                  //inicializa o LCD
     cria_novos_caract();               //cria os 8 novos caracteres

     //escreve os números 0, 1, 2, 3, 4 e 5
     escreve_BIG(0x80,0);
     escreve_BIG(0x82,1);
     escreve_BIG(0x85,2);
     escreve_BIG(0x88,3);
     escreve_BIG(0x8B,4);
     escreve_BIG(0x8E,5);

     for(;;);  //laço infinito
}
//=====================================================================================
```

Na fig. 5.29, é apresentado o resultado para o programa supracitado. O detalhe está em escolher a posição correta para a escrita do número grande.

Fig. 5.29 – Números grandes num LCD 16 × 2.

Exercícios:

5.18 – Desenvolva um programa para realizar uma animação sequencial na linha superior e inferior de um LCD 16×2. Escreva um caractere por vez.

5.19 – Crie um caça-níquel eletrônico empregando 3 caracteres diferentes apresentados em 3 posições do LCD. Utilize um botão no pino PB2 do ATmega para o sorteio.

5.20 – Crie oito caracteres novos para o LCD 16×2. Comece a criar seu próprio conjunto de funções.

5.21 – Usando as duas linhas do LCD, crie um cronômetro com passo de 1 s. Utilize os números grandes (4 caracteres por digito) e o pino PB2 do ATmega para o início/parada. Essa ideia é exemplificada na fig. 5.30.

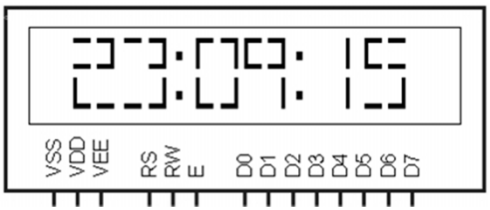


Fig. 5.30 – Números grandes para um cronômetro.

5.6 ROTINAS DE DECODIFICAÇÃO PARA USO EM DISPLAYS

Na programação em C, as variáveis podem ser de 8, 16 ou mais bits. Para apresentar em um *display* o valor de alguma dessas variáveis, é necessário decodificar o número representado pela variável em seus dígitos individuais. Por exemplo, supondo uma variável de 16 bits com o valor de 14569, como apresentar os dígitos 1, 4, 5, 6 e 9 em um LCD 16×2 ou em um conjunto de 5 *displays* de 7 segmentos?

A solução para o problema é dividir o número por 10, guardar o resto, pegar o número inteiro resultante e seguir com o processo até que a divisão resulte zero (ver os exemplos no capítulo 4). Os restos da divisão são os dígitos individuais do número (base decimal). A sub-rotina abaixo exemplifica o processo e serve para converter um número de 16 bits, sem sinal, nos seus cinco digitais individuais (valor máximo de 65.535).

```
//-------------------------------------------------------------------------------
#define tam_vetor   5 //número de digitos individuais para a conversão por ident_num()
#define conv_ascii 48 //48 se ident_num() deve retornar um número no formato ASCII (0 - normal)

unsigned char digitos[tam_vetor];//declaração da variável para armazenagem dos digitos
//-------------------------------------------------------------------------------
void ident_num(unsigned int valor, unsigned char *disp)
{
     unsigned char n;

     for(n=0; n<tam_vetor; n++)
          disp[n] = 0 + conv_ascii;      //limpa vetor para armazenagem dos digitos
     do
     {
       *disp = (valor%10) + conv_ascii;  //pega o resto da divisão por 10
        valor /=10;                      //pega o inteiro da divisão por 10
        disp++;

     }while (valor!=0);
}
//-------------------------------------------------------------------------------
```

A função recebe dois parâmetros: um deles é o **valor** a ser convertido, limitado ao tamanho da variável declarada, no caso acima, 16 bits (*unsigned int*); o outro parâmetro é o ponteiro para o vetor (***disp**) que conterá os valores individuais dos dígitos da conversão. No início da função, o vetor é zerado para garantir que valores anteriormente

convertidos não prejudiquem a conversão atual. Esse vetor deve ser declarado no corpo do programa onde será utilizado e seu tamanho é determinado pelo máximo valor que poderá conter.

O LCD com o controlador HD44780 entende somente caracteres no formato ASCII. Para ser impresso, o dado deve estar neste formato. Assim, para a conversão adequada, um número de base decimal necessita ser somado a 48 (0x30). Por exemplo, para apresentar o número 5 em uma posição do LCD, deve ser enviado o número 53 (conforme tab. B.4 do apêndice B). Em um *display* de 7 segmentos, é necessário utilizar a tabela de decodificação (tab. 5.2), como explicado na seção 5.4.

Abaixo, é apresentado um programa (baseado na seção 5.5.2) para a impressão de números de 0 a 100 em um LCD 16 × 2 empregando a função **ident_num(...)**.

LCD_4bits_ident_num.c

```
//=============================================================================== //
//        ACIONANDO UM DISPLAY DE CRISTAL LIQUIDO DE 16x2                        //
//        Uso da função ident_num(...)                                           //
//=============================================================================== //
#include "def_principais.h"    //inclusão do arquivo com as principais definições
#include "LCD.h"
//------------------------------------------------------------------------------
int main()
{
     unsigned char digitos[tam_vetor];//declaração da variável para armazenagem dos digitos
     unsigned char cont;

     DDRD = 0xFF;                          //PORTD como saída
     DDRB = 0xFF;

     inic_LCD_4bits();                     //inicializa o LCD

     while(1)
     {
        for(cont=0; cont<101; cont++)
        {
          ident_num(cont,digitos);
          cmd_LCD(0x8D,0);//desloca o cursor para que os 3 digitos fiquem a direita do LCD
          cmd_LCD(digitos[2],1);
          cmd_LCD(digitos[1],1);
          cmd_LCD(digitos[0],1);
          _delay_ms(200);                  //tempo para troca de valor
        }
     }
}
//==============================================================================
```

Dada a importância da estruturação na programação a função **ident_num(...)** foi agregada ao arquivo de funções LCD.c.

Outra conversão importante é transformar um número decimal em seus dígitos hexadecimais. Como os números são binários por natureza, basta isolar seus *nibbles*, onde cada conjunto de 4 bits é o próprio número hexadecimal, como exemplificado no código abaixo. Se o número a ser convertido tiver sinal ou não for inteiro, é preciso considerar esses fatores para realizar a conversão.

```
//-------------------------------------------------------------------
//      Decodificando um número decimal em seus dígitos hexadecimais
//-------------------------------------------------------------------
unsigned char num = 113;     //0x71
unsigned char digito1, digito0;
...
digito0 = num & 0b00001111;  /*mascara os 4 bits mais
                             significativos, resultando no
                             primeiro dígito (digito0 = 1)*/
digito1 = num >> 4;          /*desloca o valor do número 4 bits
                             para a direita,
                             resultando no segundo dígito (4 mais
                             significativos), digito1 = 7 */
...
//-------------------------------------------------------------------
```

Exercícios:

5.22 – Crie um programa que conte até 10.000 apresentando a contagem decimal em um LCD 16×2. Utilize um tempo de 100 ms para o incremento dos números, os dígitos devem ficar à direita do LCD e o número zero na frente do digito mais significativo não deve ser apresentado.

5.23 – Repita o programa acima, só que desta vez, para apresentar um número hexadecimal em uma contagem até 0x3FF.

5.24 – Conte o número de vezes que um botão foi pressionado e apresente o valor em um LCD 16 × 2. O botão pode ser conectado ao pino PB2 do ATmega e o circuito do LCD pode ser o mesmo da fig. 5.23a.
