Trabalho prático N.º 4

Objetivos

- Configurar e usar os portos de I/O do PIC32 em linguagem C.
- Implementar um sistema de visualização com dois *displays* de 7 segmentos.

Introdução

A configuração e utilização dos portos de I/O do PIC32 em linguagem C fica bastante facilitada se se utilizarem estruturas de dados para a definição de cada um dos bits dos registos a que se pode aceder. A título de exemplo, para o registo TRIS associado ao porto E, pode ser declarada uma estrutura com 8 campos (o número de bits real do porto E no PIC32MX795F512H), cada um deles com uma dimensão de 1 bit[4]:

```
struct {
  unsigned int TRISE0 : 1;      // 1-bit field (least significant bit)
  unsigned int TRISE1 : 1;      // ...
  unsigned int TRISE2 : 1;      // ...
  unsigned int TRISE3 : 1;      // ...
  unsigned int TRISE4 : 1;      // ...
  unsigned int TRISE5 : 1;      // ...
  unsigned int TRISE6 : 1;      // ...
  unsigned int TRISE7 : 1;      // 1-bit field (most significant bit)
} __TRISEbits_t;
```

A partir desta declaração pode ser criada uma instância da estrutura, por exemplo, **`TRISEbits`**:

```
__TRISEbits_t TRISEbits;
```

O acesso a um bit específico da estrutura pode então ser feito através do nome da instância seguido do nome do membro (separados pelo caracter "."). Por exemplo, a configuração dos bits 2 e 5 do port E (RE2 e RE5) como entrada e saída, respetivamente, pode ser feita com as duas seguintes instruções em linguagem C:

```
TRISEbits.TRISE2 = 1;   // RE2 configured as input
TRISEbits.TRISE5 = 0;   // RE5 configured as output
```

Seguindo esta metodologia, podem ser declaradas estruturas que representem todos os elementos que permitem a escrita, a leitura e a configuração de um porto. Tomando ainda como exemplo o porto E, para além do registo TRIS, temos ainda os registos LAT (constituído pelos bits LATE7 a LATE0) e PORT (constituído pelos bits RE7 a RE0):

```
struct {
  unsigned int RE0 : 1;
  unsigned int RE1 : 1;
  unsigned int RE2 : 1;
  unsigned int RE3 : 1;
  unsigned int RE4 : 1;
  unsigned int RE5 : 1;
  unsigned int RE6 : 1;
  unsigned int RE7 : 1;
} __PORTEbits_t;
```

```
struct {
  unsigned int LATE0 : 1;
  unsigned int LATE1 : 1;
  unsigned int LATE2 : 1;
  unsigned int LATE3 : 1;
  unsigned int LATE4 : 1;
  unsigned int LATE5 : 1;
  unsigned int LATE6 : 1;
  unsigned int LATE7 : 1;
} __LATEbits_t;
```

Sendo a instanciação destas estruturas, por exemplo:

```
__PORTEbits_t PORTEbits;
__LATEbits_t  LATEbits;
```

[4] A forma como as estruturas de dados, que definem campos do tipo bit, são declaradas depende do compilador usado. A que apresentamos é a adequada para o compilador (gcc) usado nas aulas práticas.

Do mesmo modo que se fez anteriormente para o registo TRISE, pode referenciar-se, de forma isolada, um porto de 1 bit, usando a instância PORTEbits para os bits configurados como entrada ou LATEbits para os bits configurados como saída (ver explicação mais abaixo). Por exemplo, para a atribuição à variável **abc** do valor do bit 2 do porto E pode fazer-se:

```
abc = PORTEbits.RE2;
```

A Figura 5 apresenta o diagrama de blocos de um porto de I/O de 1 bit no PIC32. Nesse esquema, já abordado nas aulas teóricas, convém destacar os dois flip-flops S1 e S2 presentes no caminho do porto para efeitos de leitura. Esses *flip-flops*, em conjunto, formam um *shift register* de duas posições que visa a sincronização do sinal externo que se pretende ler com o instante de leitura do CPU. Estes dois *flip-flops* impõem um atraso de, até, dois ciclos de relógio na propagação do sinal externo até ao barramento de dados do CPU ("data line").

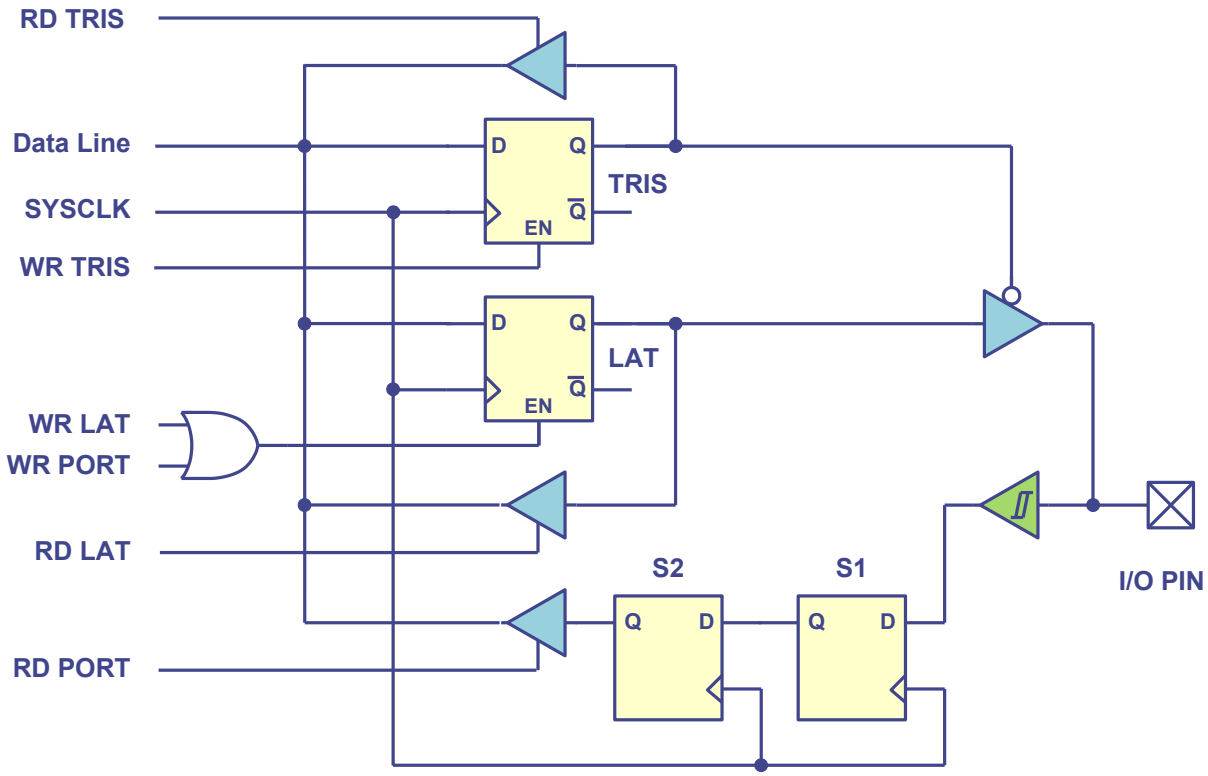

Figura 5. Diagrama de blocos simplificado de um porto de I/O no PIC32.

Numa situação em que o porto esteja configurado como saída, este atraso impõe alguns cuidados na forma como se escreve o código. Vejamos o seguinte exemplo (que pressupõe que o porto RE0 já está devidamente configurado como saída):

```
lw    $t0,PORTE($a0) # RD PORT
ori   $t0,0x0001
sw    $t0,PORTE($a0) # (RE0 = 1)
...                  # duas ou mais instruções
lw    $t0,PORTE($a0) # RD PORT
andi  $t0,0xFFFE
sw    $t0,PORTE($a0) # (RE0 = 0)
lw    $t1,PORTE($a0) # RD PORT (lê o valor antigo!)
```

Esta sequência de código escreve o valor 1 no porto RE0, a seguir escreve o valor 0 e, finalmente, lê o valor do porto RE0 para o registo $t1. O valor lido para o registo $t1 deveria ser 0 (i.e., o último valor escrito em RE0), mas será 1, ou seja, o valor que o porto apresentava antes da última operação de escrita.

Para que a última leitura do porto produza o resultado esperado, é necessário compensar o atraso, de dois ciclos de relógio, introduzido pelo *shift-register*, constituido pelos *flip-flops* S1 e S2, na leitura do valor presente no "I/O pin" (não esquecer que o MIPS inicia a execução de uma nova instrução a cada ciclo de relógio). Ou seja, é necessário separar operações consecutivas de escrita e de leitura do porto de dois ciclos de relógio, o que pode ser feito através da introdução de duas instruções **nop**, tal como se apresenta de seguida:

```
lw    $t0,PORTE($a0) # RD PORT
ori   $t0,0x0001
sw    $t0,PORTE($a0) # (RE0 = 1)
...                  # duas ou mais instruções
lw    $t0,PORTE($a0) # RD PORT
andi  $t0,0xFFFE
sw    $t0,PORTE($a0) # (RE0 = 0)
nop
nop
lw    $t1,PORTE($a0) # RD PORT
```

A alternativa à introdução das duas instruções **nop** é usar o registo LAT para a manipulação dos portos configurados como saída. Nesse caso o código ficaria:

```
lw    $t0,LATE($a0)  # RD LAT
ori   $t0,0x0001
sw    $t0,LATE($a0)  # RE0 = 1
...                  # zero ou mais instruções
lw    $t0,LATE($a0)  # RD LAT
andi  $t0,0xFFFE
sw    $t0,LATE($a0)  # RE0 = 0
lw    $t1,LATE($a0)  # RD LAT (o bit 0 de $t1 é 0)
```

Esta solução funciona porque o valor escrito em LATE num ciclo de relógio fica disponível para ser lido no ciclo de relógio seguinte, como se pode facilmente verificar no esquema da Figura 5.

Quando a programação é feita em linguagem C, e uma vez que o programador não controla a forma como o código é gerado, devem sempre usar-se os registos LATx para a manipulação dos valores em portos de saída (através de uma instância **LATxbits**). Por exemplo, a leitura do valor do bit 2 do porto E (RE2) e a sua escrita no bit 5 do mesmo porto deve ser feita, em linguagem C, do seguinte modo:

```
LATEbits.LATE5 = PORTEbits.RE2;
```

As declarações de todas as estruturas, bem como as respetivas instanciações, estão já efetuadas no ficheiro "**p32mx795f512h.h**" que é automaticamente incluido pelo ficheiro "**detpic32.h**". Logo, este último ficheiro deve ser incluído em todos os programas a escrever em linguagem C para a placa DETPIC32. Nesse ficheiro estão declaradas estruturas de dados para todos os registos de todos os portos do PIC32, bem como para todos os registos de todos os outros periféricos. Estão também feitas as necessárias associações entre os nomes das estruturas de dados que representam esses registos e os respectivos endereços de acesso. Lá também é dito que a frequência do *core* MIPS da placa DETPIC32 é 40MHz:

```
#define FREQ 40000000
```

É boa prática de programação usar o símbolo **FREQ** em vez de chapar 40000000 (ou usar **FREQ/2** em vez de 20000000) directamente no código C. Deste modo bastará recompilar o código se algum dia a frequência do *core* for alterada (a frequência máxima possível é 80MHz). O símbolo **PBCLK** (que é igual a **FREQ/2**), também está disponível.

Exemplo de programa para fazer o *toggle* do bit 0 do porto D (porto ao qual está ligado um LED na placa DETPIC32) a uma frequência de 1 Hz:

```
#include <detpic32.h>

void main(void)
{
  TRISDbits.TRISD0 = 0; // RD0 configured as output
  while(1)
  {
    while(readCoreTimer() < (FREQ/4)); // half period = 0.5s
    resetCoreTimer();
    LATDbits.LATD0 = !LATDbits.LATD0;
  }
}
```

A forma como as estruturas de dados para cada registo estão organizadas permite também o acesso a um dado registo como se se tratasse de uma variável de tipo inteiro, i.e., 32 bits (a descrição da estrutura feita acima não contempla esta possibilidade). Por exemplo, a configuração dos portos RE3 a RE1 como saída, e do porto RE0 como entrada pode fazer-se do seguinte modo:

```
TRISE = (TRISE & 0xFFF1) | 0x0001;  // RE3 to RE1 configured as outputs
                                    // RE0 configured as input
```

Do mesmo modo, se se pretender alterar os portos RE3 e RE2, colocando-os a 1 e 0, respetivamente, sem alterar o valor de RE1, pode fazer-se:

```
LATE = (LATE & 0xFFF3) | 0x0008;    // RE3=1; RE2=0;
```

Notas importantes:

- A escrita num porto configurado como entrada não tem qualquer consequência. O valor é escrito na *latch* do porto mas não fica disponível no exterior uma vez que é barrado pela porta *tri-state* que se encontra na saída da *latch* e que está em alta impedância (ver Figura 5).
- A configuração como saída de um porto que deveria estar configurado como entrada (e que tem um dispositivo de entrada associado) pode destruir esse porto. É, assim, muito importante que a configuração dos portos seja feita com grande cuidado.
- Após um *reset* (ou após *power-up*) os portos do PIC32 ficam todos configurados como entradas.
- O compilador gcc também permite especificar constantes em binário, usando o prefixo 0b. Por exemplo, 0x13 é o mesmo que 0b10011. Em alguns casos, especificar as contantes em binário (desde que não tenham muitos bits!) pode tornar o programa mais fácil de entender.

Trabalho a realizar[5]

1. Refaça, como trabalho de casa de preparação para esta trabalho prático, o terceiro trabalho prático, utilizando linguagem C de acordo com a descrição feita na secção anterior. Tome atenção, em particular, à função **`delay()`**, uma vez que esta será usada extensivamente neste trabalho prático.

2. Monte os 2 *displays* de 7 segmentos e os dois transístores, de acordo com o esquema da Figura 6 (mantenha os LEDs montados na aula anterior). Verifique no *datasheet* do *display* de 7 segmentos (disponível no site de AC2) qual a correspondência entre cada segmento e o respetivo pino físico. Para o transístor, verifique quais os pinos físicos correspondentes à *gate* (G), *drain* (D) e *source* (S) (note que a montagem incorrecta deste componente impede, ou limita, o correto funcionamento do circuito de visualização).

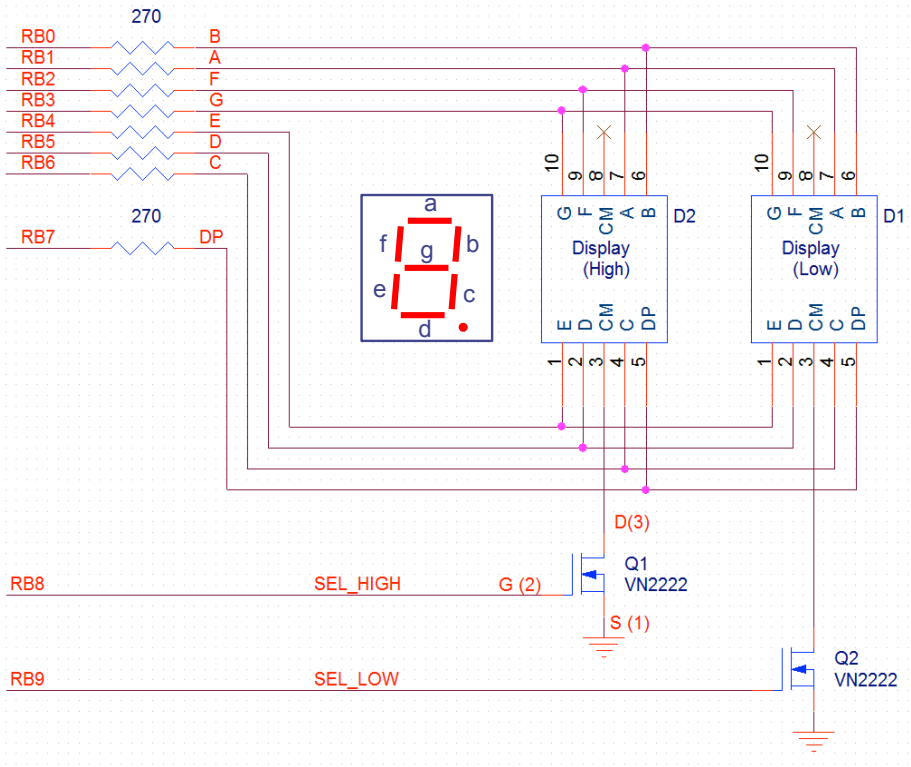

Figura 6. Ligação de dois *displays* de 7 segmentos ao porto B do PIC32.

3. Faça um programa que configure os portos RB0 a RB9 como saídas e que verifique o bom funcionamento da montagem feita no ponto 2. Para isso o programa deve selecionar apenas o "display low" (RB9=1, i.e. "**`SEL_LOW`**"=1, e RB8=0) e, em ciclo infinito, deve executar os seguintes passos:

   a) ler um caracter do teclado e esperar que seja digitada uma letra entre 'a' e 'g' (ou 'A' e 'G') ou o caracter '.' . Use o system call **`getChar()`**;

   b) escrever no porto B a combinação binária que active (apenas) o segmento do *display* correspondente ao caracter lido.

   Teste o programa para todos os segmentos e repita o procedimento para o "display high" (RB9=0 e RB8=1).

[5] **Note bem**: a organização da componente prática de AC2 pressupõe que todo o trabalho de montagem dos circuitos seja realizado fora da aula prática. Este aspeto será tido em consideração para efeitos de cálculo da nota respeitante à avaliação do docente.

4. Selecionando em sequência o "display low" e o "display high" envie para os portos RB0 a RB7, em ciclo infinito e com uma frequência de 1 Hz, a sequência binária que ativa os segmentos do *display* pela ordem a, b, c, d, e, f, g, a, ...; o período de 1s deve ser obtido através da função **`delay()`** que implementou no ponto 1 deste trabalho prático.

```
void main(void)
{
   static const unsigned char codes[] = {0x02, 0x.., ...};
   // configure RB0-RB9 as outputs
   //
   while(1)
   {
      // select display low
      for(i=0; i < 7; i++)
      {
         // write codes[i] in port B
         // wait 1 second
      }
      // select display high
      ...
   }
}
```

5. Construa a tabela que relaciona as combinações binárias de 4 bits (dígitos 0 a F) com o respetivo código de 7 segmentos, de acordo com o circuito montado no ponto anterior e com a definição gráfica dos dígitos apresentada na Figura 7.

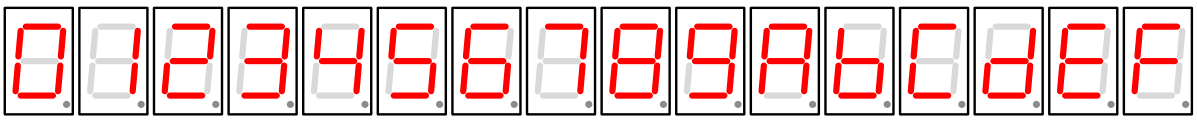

Figura 7. Representação dos dígitos de 0 a F no *display* de 7 segmentos.

6. Complete a montagem do *dip-switch* que fez no terceiro trabalho prático, de acordo com o seguinte esquema (em vez de fazer ligações diretas a 3.3V, pode manter as resistências de 470Ω que já tinha, e utilizar outras para as ligações adicionais a +3.3V):

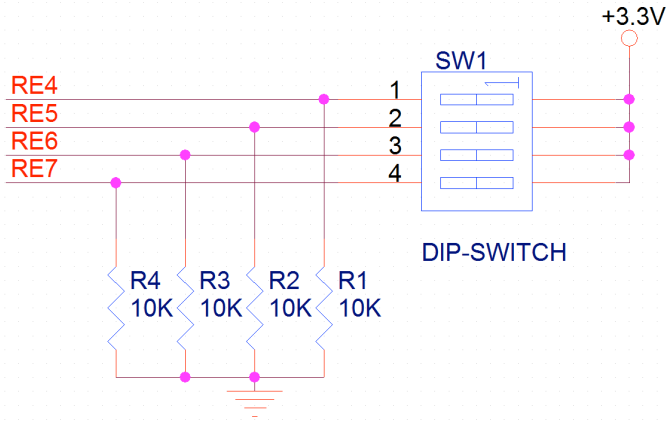

Figura 8. *Dip-switch* de 4 posições ligado a 4 bits do porto E.

7. Escreva um programa que leia o valor do *dip-switch* (4 bits), faça a conversão para o código de 7 segmentos respetivo e escreva o resultado no *display* menos significativo (não se esqueça de configurar previamente os portos RE4 a RE7 como entradas). Repita o procedimento para a escrita no *display* mais significativo.

```
void main(void)
{
   static const unsigned char display7Scodes[] = {0x.., 0x.., ...};
   // configure RE4 to RE7 as inputs
   while(1)
   {
      // read dip-switch
      // convert to 7 segments code
      // send to display
   }
}
```

8. Escreva uma função que envie um byte hexadecimal (2 dígitos) para os *displays*:

```
void send2displays(unsigned char value)
{
   static const unsigned char display7Scodes[] = {0x.., 0x.., ...};
   // send digit_low to diplay_low
   // send digit_high to diplay_high
}
```

9. Escreva um programa que implemente um contador binário de 8 bits. O contador deve ser incrementado com uma frequência de 5 Hz e o seu valor deve ser enviado, ao mesmo ritmo, para os *displays* através da função **`send2displays()`** escrita no ponto anterior. Utilize a função **`delay()`** para gerar um atraso de 200 ms e dessa forma determinar a frequência de incremento/visualização.

10. Como pode observar, o sistema de visualização apresenta um comportamento bastante deficiente. Com a configuração usada, é necessário enviar de forma alternada os valores para os dois *displays*. Se o tempo de ativação de cada um dos dois *displays* não for o mesmo, o brilho exibido por cada um deles será também diferente.

    Por outro lado, será necessário aumentar a frequência de trabalho do processo de visualização de modo a que o olho humano não detecte as alternâncias na seleção dos *displays*. Assim, de modo a melhorar o desempenho do sistema de visualização, teremos que 1) garantir que o tempo de ativação dos dois *displays* é o mesmo e 2) aumentar a frequência de refrescamento do sistema de visualização.

    a. Reescreva a função **`send2displays()`** de modo a que sempre que for invocada envie apenas um dos dois dígitos, de forma alternada, para o sistema de visualização.

```
void send2displays(unsigned char value)
{
   static unsigned char displayFlag = 0;
   static const unsigned char display7Scodes[] = {0x.., 0x.., ...};

   digit_low = value & 0x0F;
   digit_high = value >> 4;
   // if "displayFlag" is 0 then send digit_low to diplay_low
   // else send digit_high to diplay_high
   // toggle "displayFlag" variable
}
```

    b. Reescreva o programa principal, tal como se esquematiza abaixo, de modo a invocar a função **`send2displays()`** com uma frequência de 20 Hz, continuando a usar a função **`delay()`** para determinar as frequências de visualização (20 Hz) e de contagem (5 Hz).

```
void main(void)
{
   // declare variables
   // initialize ports
   while(1)
   {
      i = 0;
      do
      {
         // wait 50 ms
         // call send2displays with counter value as argument
      } while(++i < 4);
      // increment counter (module 256)
   }
}
```

11. Com as alterações introduzidas no ponto anterior, o brilho de cada um dos dois *displays* ficou equilibrado. Continua, contudo, a notar-se a comutação entre os dois *displays,* efeito que é comum designar-se por *flicker*. De modo a diminuir, ou mesmo eliminar, o *flicker*, a frequência de refrescamento (*refresh rate*) tem que ser aumentada (no programa anterior era efetuado um refrescamento a cada 50 ms, isto é, a uma frequência de 20 Hz).

    Assim, mantendo a frequência de actualização do contador em 5Hz, altere o programa anterior de forma a aumentar a frequência de refrescamento para 50 Hz (20 ms) e depois para 100 Hz (10 ms). Observe os resultados num e noutro caso.

12. Utilize o osciloscópio para visualizar os dois sinais de selecção dos *displays* ("**SEL_HIGH**" e "**SEL_LOW**"). Meça o tempo de ativação desses sinais para as frequências de refrescamento de 50 e 100 Hz.

13. Mantendo a frequência de refrescamento em 100 Hz, altere o programa anterior de modo a incrementar o contador em módulo 60. A frequência de incremento deverá ser 1 Hz e os valores devem ser mostrados em decimal. A conversão para decimal pode ser feita, de forma simplificada e desde que o valor de entrada seja representável em decimal com dois dígitos, pela seguinte função:

```
unsigned char toBcd(unsigned char value)
{
   return ((value / 10) << 4) + (value % 10);
}
```

14. Acrescente ao programa anterior o controlo do ponto decimal dos *displays*, de modo a que quando o valor do contador for par fique ativo o ponto das unidades e quando for ímpar fique ativo o das dezenas.
15. Altere o programa anterior de modo a que quando a contagem dá a volta (isto é, quando o valor do contador volta a zero), o valor 00 fique a piscar (meio segundo *on*, meio segundo *off*) durante 5 segundos, antes da contagem ser retomada.

Elementos de apoio

- Slides das aulas teóricas.
- *Data sheets* dos circuitos do *display* e do transístor VN2222 (disponíveis no site de AC2).