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. Por exemplo, para o registo **TRIS** associado ao porto **RE**, pode ser declarada uma estrutura com 8 campos (o número de bits real do porto **RE** no PIC32MX795F512H), cada um deles com a dimensão de 1 bit[1]:

```
typedef 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; // TRISEbits é uma instância de __TRISEbits_t
```

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 "." (e.g. **TRISEbits.TRISE7**). Por exemplo, a configuração dos bits 2 e 5 do porto **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 registos necessários para a leitura, a escrita 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**):

```
typedef 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;
```

```
typedef 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:

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

[1] A forma como são declaradas as estruturas de dados que definem campos do tipo bit depende do compilador usado (a que se apresenta é a usada pelo compilador usado nas aulas práticas, pic32-gcc).

Do mesmo modo que se fez anteriormente para o registo **TRISE**, pode-se referenciar, de forma isolada, um porto I/O de 1 bit: usando a instância **PORTEbits** pode-se ler o valor de um porto de entrada; usando a instância **LATEbits** pode-se aceder ao flip-flop **LAT** do porto **E**, para ler ou para escrever. Por exemplo, para atribuir ao porto de saída **RB4**, o valor presente no porto de entrada **RE2** pode-se fazer:

```
LATBbits.LATB4 = PORTEbits.RE2; // atribui ao porto RB4 o valor lido do
                                // porto RE2
```

Na tradução para *assembly*, o compilador gera sequências do tipo "Read/Modify/Write", de modo a preservar o valor dos bits que não se pretendem alterar. Para o exemplo anterior, o compilador produz, tipicamente, a seguinte sequência de instruções (admitindo que o registo **$t0** já foi inicializado com os 16 bits mais significativos do endereço dos portos):

```
lw    $t2, PORTE($t0)
andi  $t2, $t2, 0x0004
sll   $t2, $t2, 2
lw    $t3, LATB($t0)
andi  $t3, $t3, 0x0FFEF
or    $t3, $t3, $t2
sw    $t3, LATB($t0)
```

A Figura 1 apresenta o diagrama de blocos de um porto de I/O de 1 bit no PIC32. Nesse esquema, para além dos registos **TRIS** e **LAT**, destacam-se ainda os dois flip-flops S1 e S2 presentes no caminho do porto para efeitos de leitura. Esses *flip-flops*, em conjunto, formam um circuito sincronizador que visa resolver os possíveis problemas causados por meta-estabilidade decorrentes do facto de o sinal externo ser assíncrono relativamente ao *clock* 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").

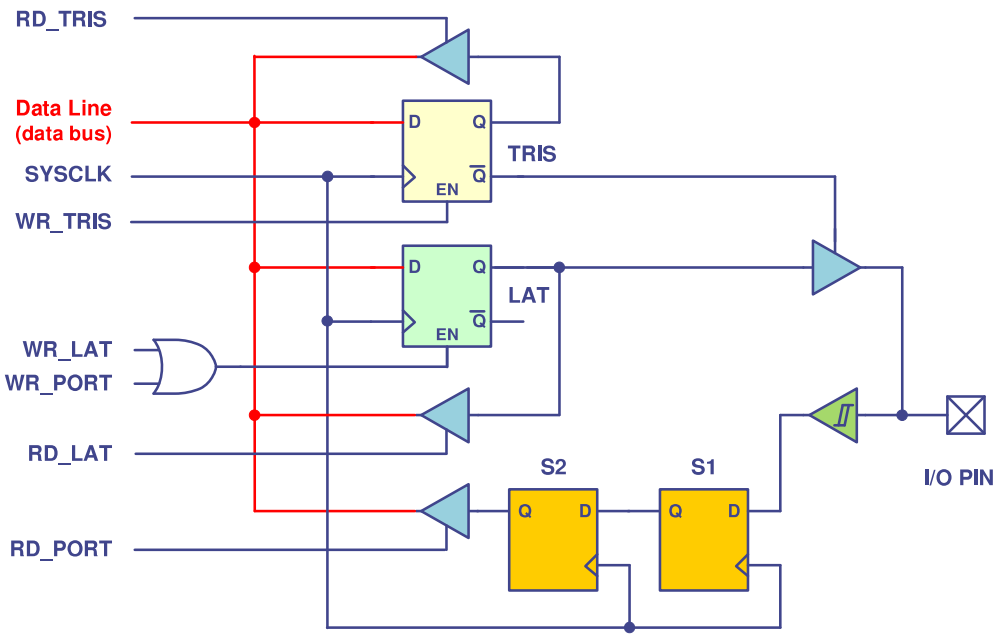

Figura 1. Diagrama de blocos simplificado de um porto de I/O no PIC32.

Para a manipulação dos valores a enviar para os portos configurados como saída devem sempre usar-se os registos **LATx** (ver explicação fornecida em anexo).

Exemplos:

a) Atribuição do valor '1' ao bit 3 do porto B:

```
LATBbits.LATB3 = 1;
```

b) Leitura do porto **RE2** (bit 2 do porto E) e escrita do seu valor, negado, no bit 5 do porto B:

```
LATBbits.LATB5 = !PORTEbits.RE2;
```

c) Inversão do valor de um porto de saída (por exemplo bit 0 do porto D):

```
LATDbits.LATD0 = !LATDbits.LATD0;
```

A forma como as estruturas de dados estão organizadas permite também o acesso a um dado registo (para ler ou escrever) tratando-o como 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-se fazer do seguinte modo:

```
TRISE = (TRISE & 0xFFF0) | 0x0001; // RE3 a RE1 configurados como saídas
                                   // RE0 configurado como entrada
```

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** (nem qualquer outro dos restantes), pode-se fazer[2]:

```
LATE = (LATE & 0xFFF3) | 0x0008;   // RE3=1; RE2=0; RE1 mantém o valor
```

Ficheiro detpic32.h

As declarações de todas as estruturas, bem como as respetivas instanciações, estão já feitas no ficheiro "**p32mx795f512h.h**" disponibilizado pelo fabricante, que é automaticamente incluído 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 respetivos endereços de acesso.

Está igualmente definida no ficheiro "**detpic32.h**" a frequência de funcionamento do *core* MIPS da placa DETPIC32 (previamente configurada para 40MHz):

```
#define FREQ 40000000   // 40 MHz
```

É boa prática de programação usar o símbolo **FREQ** em vez de usar a constante **40000000** (ou usar **FREQ/2** em vez de **20000000**) diretamente no código C. Deste modo bastará recompilar o código se algum dia a frequência do *core* for alterada (a frequência máxima possível, na versão usada na placa DETPIC32, é 80MHz). O símbolo **PBCLK** (que é igual a **FREQ/2**), também está definido:

```
#define PBCLK  (FREQ / 2)
```

[2] O compilador gcc 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.

Notas importantes:

- A escrita num porto configurado como entrada não tem qualquer consequência. O valor é escrito no *flip-flop* LAT associado ao porto mas não fica disponível no exterior uma vez que é barrado pela porta *tri-state* que se encontra na saída e que está em alta impedância (ver Figura 1).
- A configuração como saída de um porto que deveria estar configurado como entrada (e que tem um dispositivo de entrada associado) pode, em algumas circunstâncias, 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.