universidade de aveiro theoria poiesis praxis

Departamento de Electrónica, Telecomunicações e Informática

Arquitectura de Computadores II

Mestrado Integrado em Engenharia Electrónica e Telecomunicações

Mestrado Integrado em Engenharia de Computadores e Telemática

2º Ano, 2º semestre

Guia das Aulas Práticas

Ano letivo 2013/2014

(janeiro/2014)

José Luís Azevedo, Arnaldo Oliveira, Tomás Oliveira e Silva

Trabalho prático N.º 1

Objetivos

- Conhecer o processo de criação de um programa escrito em *assembly* para correr na placa DETPIC32: compilação, transferência e execução.
- Utilizar os *system calls* disponibilizados na placa DETPIC32.
- Rever os conceitos associados à manipulação de *arrays* de caracteres.

Trabalho a realizar

Parte I

1. Utilizando o editor de texto *gvim*[1], edite e grave o programa de demonstração *assembly* que é apresentado de seguida. Para facilitar a organização dos ficheiros dos vários programas que irão ser feitos ao longo do semestre, sugere-se que seja criado um *folder* (diretório) por trabalho prático, estando o nome do ficheiro relacionado com a alínea a que diz respeito. No nosso caso o ficheiro poder-se-á chamar "prog1.s" (usa-se a extensão ".s" para ficheiros *assembly*) a colocar no *folder* "tp01".

```
#          int main(void)
#          {
#             printStr("AC2 – DETPIC32 primer\n");   // system call
#             return 0;
#          }

           .equ   PRINT_STR,8

           .data
msg:       .asciiz "AC2 - DETPIC32 primer\n"

           .text
           .globl  main
main:      la     $a0,msg
           ori    $v0,$0,PRINT_STR
           syscall                 # printStr("AC2 – DETPIC32 primer\n");
           ori    $v0,$0,0         # return 0;
           jr     $ra
```

2. Compile o programa do ponto 1 introduzindo, na linha de comando (numa janela de terminal do linux), o seguinte comando**:**

   `pcompile prog1.s`

3. O comando da linha anterior produz os seguintes ficheiros: "prog1.o", "prog1.elf", "prog1.map" e "prog1.hex", sendo os dois primeiros ficheiros binários e os restantes de texto. Analise o conteúdo do ficheiro "prog1.hex" das duas formas seguintes:
   - utilizando o editor de texto *gvim*
   - utilizando o *disassembler hex2asm*, com as seguintes linhas de comando:

     `hex2asm prog1.hex` (gera o ficheiro "prog1.hex.s")

     `gvim prog1.hex.s`

   - identifique no ficheiro "prog1.hex.s" os endereços correspondentes aos *labels* **`msg`** e **`main`** do programa que editou. Anote no seu *log book* esses endereços.

[1] Embora possam ser usados outros editores de texto, o *gvim* é o recomendado por questões de portabilidade e legibilidade dos ficheiros.

4. Analise o conteúdo do ficheiro "prog1.map" produzido pelo processo de compilação, abrindo-o com o editor de texto *gvim*. Compare os endereços dos *labels* **`msg`** e **`main`** com os obtidos no ponto anterior (acrescente "**`.globl msg`**" ao código anterior para o símbolo "**`msg`**" passar a aparecer no ficheiro "prog1.map").
5. Transfira o programa "prog1.hex" para a memória FLASH do microcontrolador PIC32 da placa DETPIC32, realizando os seguintes passos:
   - ligue a placa à porta USB do PC
   - introduza o comando: **`ldpic32 –w prog1.hex`**
   - prima o botão de *reset* da placa DETPIC32 e aguarde que a transferência se processe
   - execute, em linha de comando, o programa **`pterm`**
6. Execute o programa transferido no ponto anterior premindo novamente o botão de *reset*.
7. Desligue a placa DETPIC32 da porta USB do PC, espere uns segundos, volte a ligar e repita o ponto 6.

Parte II

1. Os programas que se apresentam de seguida exercitam a utilização dos *system calls* disponíveis na placa DETPIC32. Verifique os *system calls* disponibilizados, consultando ou a tabela de referência rápida referida nos elementos de apoio no final deste trabalho prático ou dando uma vista de olhos no ficheiro "/opt/pic32mx/include/detpic32.h". Analise a forma como cada um dos *system calls* deve ser invocado (por motivos históricos, os números dos *system calls* são diferentes dos usados em AC1).
2. Identifique a funcionalidade de cada um dos programas que se seguem e traduza-os para *assembly* do MIPS, usando as convenções de passagem de parâmetros e salvaguarda de registos que estudou em AC1. Compile cada um dos programas *assembly*, do mesmo modo que realizou nos pontos 2 a 5 da Parte I. Transfira o resultado da compilação (ficheiros ".hex") para a placa DETPIC32 e verifique o respetivo funcionamento.

   NOTA: O código que escrever vai ser executado numa arquitetura *pipelined* com *delayed branches*. Ou seja, em todas as instruções que alteram o fluxo de execução (**`beq`**, **`bne`**, **`j`**, **`jal`**, **`jr`**, **`jalr`**) a instrução que vem imediatamente a seguir é sempre executada, independentemente do comportamento da instrução de salto. Apesar disso, não necessita de ter em conta este comportamento, uma vez que o *assembler* efetua, de forma automática, a reordenação das instruções de modo a preencher, sempre que possível o *delayed slot*. Nos casos em que o *assembler* deteta que não pode reordenar as instruções devido a dependência(s) de dados, o *delayed slot* é preenchido com a instrução "**nop**". Este comportamento deve ser verificado através da análise do ficheiro produzido pelo programa **`hex2asm`** (por exemplo "prog1.hex.s")

```
// ************************************************************************
// Programa 2 – teste do system call "inkey()"
// ************************************************************************
int main(void)
{
   char c;

   while (1)
   {
      while ((c = inkey()) == 0);
      if (c == '\n')
         break;
      printStr("Key pressed\n");
   }
   return 0;
}
```

```
// ************************************************************************
// Programa 3 – teste dos system calls "getChar()" e "putChar()"
// ************************************************************************
int main(void)
{
   char c;

   while (1)
   {
      c = getChar();
      if (c == '\n')
         break;
      putChar(c);
   }
   return 1;
}

// ************************************************************************
// Programa 4 – teste dos system calls de leitura e impressão de inteiros
// ************************************************************************
void main(void)
{
   int value;

   while (1)
   {
      printStr("\nIntroduza um numero (sinal e módulo): ");
      value = readInt10();
      printStr("\nValor lido em base 2: ");
      printInt(value, 2);
      printStr("\nValor lido em base 16: ");
      printInt(value, 16);
      printStr("\nValor lido em base 10 (unsigned): ");
      printInt(value, 10);
      printStr("\nValor lido em base 10 (signed): ");
      printInt10(value);
   }
}

// ************************************************************************
// Programa 5 – teste do system call "readStr()" e manipulação de strings
// ************************************************************************
#define STR_MAX_SIZE 20

char *strcat(char *, char *);
char *strcpy(char *, char *);
int strlen(char *);

int main(void)
{
   static char str1[STR_MAX_SIZE + 1];
   static char str2[STR_MAX_SIZE + 1];
   static char str3[2 * STR_MAX_SIZE + 1];

   printStr("Introduza 2 strings: ");
   readStr( str1, STR_MAX_SIZE );
   readStr( str2, STR_MAX_SIZE );
   printStr("Resultados:\n");
   prinInt( strlen(str1), 10 );
   prinInt( strlen(str2), 10 );
   strcpy(str3, str1);
   printStr( strcat(str3, str2) );
   printInt10( strcmp(str1, str2) );
   return 0;
}
```

Note: a versão do *assembler* que está a ser usada nas aulas práticas não interpreta corretamente o caracter de terminação das strings, '\0'; em *assembly* use, em vez deste caracter, o valor 0 (tal como está no código abaixo).

```
// ***************************************************************************
// String length
// ***************************************************************************
//
int strlen(char *s)
{
   int len;
   for(len = 0; *s != 0; len++, s++);
   return len;
}

// ***************************************************************************
// String concatenate
// ***************************************************************************
//
char *strcat(char *dst, char *src)
{
   char *rp = dst;

   for(; *dst != 0; dst++);
   strcpy(dst, src);
   return rp;
}

// ***************************************************************************
// String copy
// ***************************************************************************
//
char *strcpy(char *dst, char *src)
{
   char *rp = dst;

   for(; (*dst = *src) != 0; dst++, src++);
   return rp;
}

// ***************************************************************************
// String compare (alphabetically).
// Returned value is:
//   < 0  string "s1" is less than string "s2"
//   = 0  string "s1" is equal to string "s2"
//   > 0  string "s1" is greater than string "s2"
// ***************************************************************************
//
int strcmp(char *s1, char *s2)
{
   for(; (*s1 == *s2) && (*s1 != 0); s1++, s2++);
   return(*s1 - *s2);
}
```

Elementos de apoio

- Tabela com resumo do conjunto de instruções da arquitectura MIPS, na versão adaptada a Arquitectura de Computadores II (disponível no site da disciplina).
- Slides das aulas teóricas de Arquitectura de Computadores I.
- David A. Patterson, John L. Hennessy, Computer Organization & Design – The Hardware/Software Interface, Morgan Kaufmann Publishers.

Trabalho prático N.º 2

Objetivos

- Consolidar os conceitos associados à implementação em *assembly* de estruturas de controlo de fluxo de execução.
- Rever os conceitos sobre a manipulação de *arrays* de inteiros com e sem sinal.

Trabalho a realizar

Parte I

1. Traduza para *assembly* do MIPS o programa em linguagem C que se apresenta na página seguinte. Esse programa preenche um *array* de inteiros, efetua a sua ordenação por ordem crescente e apresenta o resultado.
2. Verifique o correto funcionamento do programa para vários conjuntos de valores de entrada.
3. Altere a codificação do programa de ordenação de modo a fazer ordenação por ordem decrescente.
4. Reescreva o programa dado em C de modo a permitir a ordenação de inteiros com sinal e reflita essas alterações na codificação *assembly* (incluíndo as necessárias alterações nos *system calls*). Verifique o correto funcionamento do programa introduzindo, como dados de entrada, inteiros positivos e negativos.

Parte II

O *core* MIPS disponível no microcontrolador PIC32 implementa, no coprocessador 0, um contador crescente de 32 bits (designado por *core timer*) atualizado a cada dois ciclos de relógio do CPU. Na placa DETPIC32 o relógio do CPU está configurado a 40 MHz, pelo que o contador é incrementado a uma frequência de relógio de 20 MHz. Isto significa que o tempo necessário para incrementar o contador desde o valor 0 até 20.000.000 (0x01312D00) é 1 segundo.

A placa DETPIC32 disponibiliza dois *system calls* para interagir com esse contador: ler o valor atual do contador (**`readCoreTimer()`**) e reiniciar a zero o seu valor (**`resetCoreTimer()`**).

1. Usando os *system calls* adequados, escreva em C e traduza para *assembly* do MIPS um programa que permita visualizar, em ciclo infinito, o valor atual do contador.
2. Usando os *system calls* adequados, escreva em C e traduza para *assembly* do MIPS um programa que incremente uma variável (começando em 0) à frequência de 1 segundo. De cada vez que a variável for atualizada, o seu valor deve ser apresentado no ecrã. Verifique o correto funcionamento do programa.
3. Com base no ponto anterior, escreva em C e traduza para *assembly* um programa que implemente um relógio digital no formato HH:MM:SS (começando em valores pedidos ao utilizador no início do programa). Verifique o correto funcionamento do programa.

Elementos de apoio

- Tabela com resumo do conjunto de instruções da arquitectura MIPS, na versão adaptada a Arquitectura de Computadores II (disponível no site da disciplina).
- Slides das aulas teóricas de Arquitectura de Computadores I.
- David A. Patterson, John L. Hennessy, Computer Organization & Design – The Hardware/Software Interface, Morgan Kaufmann Publishers.

```
// ************************************************************************
// AC2 – Programa 1
// ************************************************************************

#define     N_INT 5
#define     TRUE  1
#define     FALSE 0

unsigned int sequentialSort(unsigned int, unsigned int *);
void troca(unsigned int *v1, unsigned int *v2);

void main(void)
{
   static unsigned int lista[N_INT];     // reservado no segmento de dados
   unsigned int i, n_trocas;
   unsigned int *ptr;

   printStr("\nLeitura e ordenacao de inteiros em base 10\n");
   printStr("Introduza 5 Inteiros: ");   // system call

   for( i = 0; i < N_INT; i++ )
   {
      lista[i] = readInt(10);            // system call
      putChar(' ');          // system call
   }

   n_trocas = sequentialSort( N_INT, lista );

   printStr("\nNumero de trocas realizado: ");
   printInt(n_trocas, 10);         // system call

   printStr("\nResultado da ordenacao: ");

   for( ptr = lista; ptr < lista + N_INT; ptr++ )
   {
      printInt(*ptr, 10);          // system call
      putChar(' ');                // system call
   }
}

unsigned int sequentialSort(unsigned int n_val, unsigned int *array)
{
   unsigned int i, j, n_trocas=0;

   for( i = 0; i < n_val - 1; i++ )
   {
      for( j = i + 1; j < n_val; j++ )
      {
         if( array[i] > array[j] )
         {
            troca(&array[i], &array[j]);
            n_trocas++;
         }
      }
   }
   return n_trocas;
}

void troca(unsigned int *v1, unsigned int *v2)
{
   unsigned int aux;
   aux = *v1;
   *v1 = *v2;
   *v2 = aux;
}
```

Trabalho prático N.º 3

Objetivos

- Conhecer a estrutura básica e o modo de configuração de um porto de I/O no microcontrolador PIC32.
- Interligar dispositivos simples de interação com o utilizador a portos de I/O do PIC32.
- Configurar em *assembly* os portos de I/O do PIC32 e aceder para enviar / receber informação do exterior.

Nota: montar placa antes da aula.

Introdução

O microcontrolador PIC32 disponibiliza vários portos de I/O, com várias dimensões (número de bits), identificados com as siglas PORTB, PORTC, PORTD, PORTE, PORTF e PORTG. Cada um dos bits de cada um destes portos pode ser configurado, por programação, como entrada ou saída. Poderemos então considerar um porto de I/O de **n** bits do PIC32 como um conjunto de **n** portos de I/O de 1 bit. Por exemplo, o bit 0 do porto E (designado por RE0) pode ser configurado como entrada e o bit 1 do mesmo porto (RE1) ser configurado como saída.

A configuração de cada um dos bits de um porto como entrada ou saída pode ser efetuada através dos registos TRISxn, em que **x** é a letra identificativa do porto e **n** o bit desse porto que se pretende configurar. Por exemplo, para configurar o bit 0 do porto E (RE0) como entrada, o bit 0 do registo TRISE deve ser colocado a 1 (i.e. TRISE0 = 1); para configurar o bit 1 do porto E (RE1) como saída, o bit 1 do registo TRISE deve ser colocado a 0 (TRISE1 = 0).

Em termos de modelo de programação, cada porto tem associados 12 registos de 32 bits (numa visão simplificada), dos quais, nesta fase, apenas usaremos 3: os registos TRISx, PORTx e LATx. Esses registos estão mapeados no espaço de endereçamento de memória (área designada por SFRs), em endereços pré-definidos disponíveis nos manuais do fabricante, pelo que o acesso para leitura e escrita é efetuado através das instruções LW e SW da arquitetura MIPS.

```
.equ SFR_BASE_HI, 0xBF88      # 16 MSbits of SFR area
.equ TRISE, 0x6100            # TRISE address is 0xBF886100
.equ PORTE, 0x6110            # PORTE address is 0xBF886110
.equ LATE,  0x6120            # LATE  address is 0xBF886120
```

A título de exemplo, se se pretender configurar, em *assembly*, os bits 0 e 3 do porto E (RE0 e RE3) como saída podemos utilizar a seguinte sequência de código:

```
lui     $t1, SFR_BASE_HI      #
lw      $t2, TRISE($t1)       # Mem_addr = 0xBF880000 + 0x6100
andi    $t2, $t2, 0xFFF6      # bit0 = bit3 = 0 (0 means OUTPUT)
sw      $t2, TRISE($t1)       # Write TRISE register
```

Para colocar as saída dos portos RE0 e RE3 a 1 pode fazer-se:

```
lui     $t1, SFR_BASE_HI      #
lw      $t2, LATE($t1)        # Read LATE register
ori     $t2, $t2, 9           # Set bit0 and bit3
sw      $t2, LATE($t1)        # Write LATE register
```

Como auxiliar de memória, note que o registo TRIS controla o estado *tri-state* do porto (0 = *tri-state off* → o porto não está no estado de alta impedância → porto de saída), PORT diz respeito ao valor no pino (como veremos mais adiante, o valor lido corresponde ao valor no pino dois ciclos de relógio atrás), e LAT diz respeito ao valor na *latch*.

Trabalho a realizar[2]

Parte I

Ligue, de acordo com o esquema seguinte, um *led* e um contacto do *dip-switch* aos bits 0 e 6 do porto E, pinos RE0 e RE6, respetivamente, da placa DETPIC32.[3]

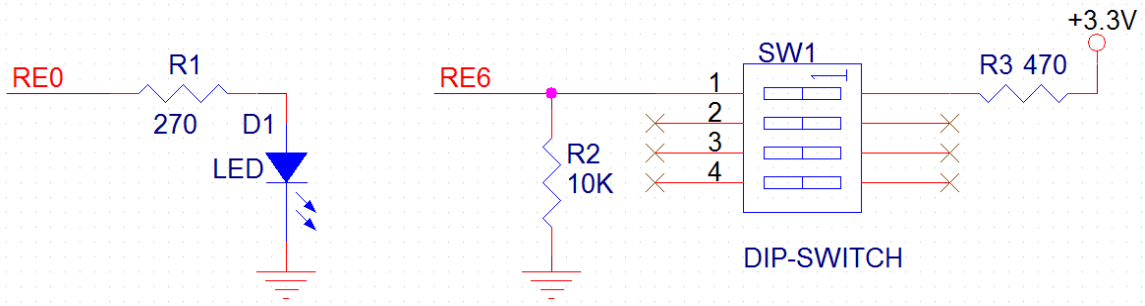


Figura 1. Ligação de um *led* e um *dip-switch* a portos do PIC32.

1. Escreva e teste um programa em *assembly* que:
   - configure os portos RE0 e RE6 como saída e entrada, respetivamente;
   - em ciclo infinito, leia o valor do porto de entrada e escreva esse valor no porto de saída (i.e RE0 = RE6).
2. Altere o programa anterior de modo a escrever no porto de saída o valor lido do porto de entrada, negado (i.e. RE0 = RE6\).
3. Ligue um segundo contacto do *dip-switch* ao porto RE7 e mais três *leds* (e as respetivas resistências), aos portos RE1, RE2 e RE3.

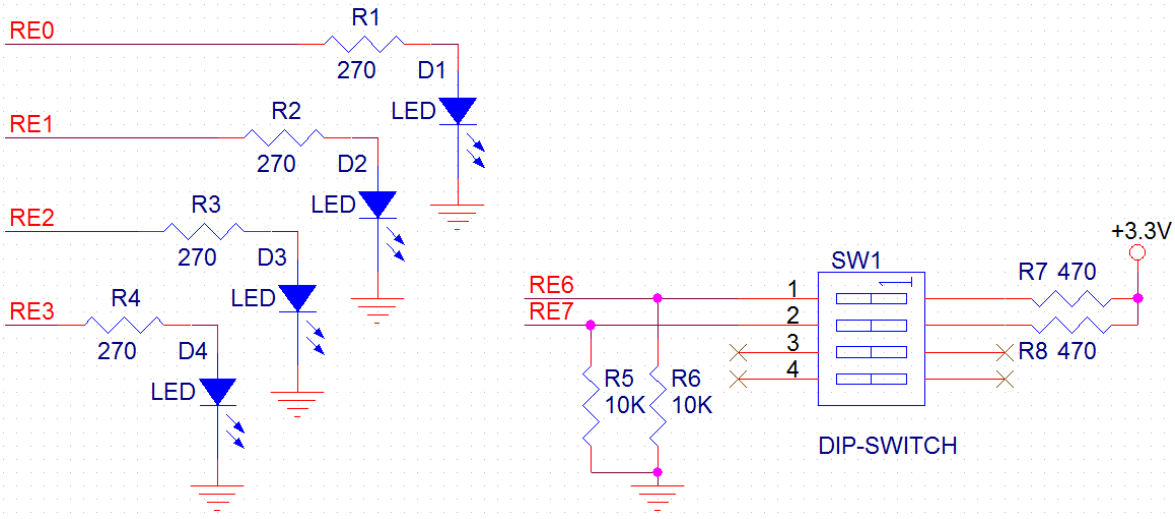


Figura 2. Ligação de 4 *leds* e um *dip-switch* de 4 posições a portos do PIC32.

4. Escreva e teste um programa em *assembly* que:
   - configure os portos RE0, RE1, RE2 e RE3 como saídas e os portos RE6 e RE7 como entradas;

[2] **Note bem**: a organização da componente prática de AC2 pressupõe que todo o trabalho de montagem dos circuitos seja efetuado fora da aula prática. (Isto aplica-se a este e aos guiões seguintes.) Este aspeto será tido em consideração para efeitos de cálculo da nota respeitante à avaliação do docente.

[3] A resistência de 10kΩ destina-se a forçar a tensão em RE6 a zero quando o *switch* está aberto. A de 470Ω foi incluida apenas por segurança, para evitar uma possível corrente muito elevada caso o porto RE6 seja configurado por engano como uma saída (quando o *switch* está fechado com o porto configurado como uma entrada, a tensão em RE6 estará perto de 3.3V). Em ambos os casos, é possível utilizar resistências com outros valores (digamos, entre 10kΩ e 33kΩ, e entre 470Ω e 1kΩ).

- em ciclo infinito, leia os valores dos portos de entrada e escreva nos portos de saída os resultados das seguintes expressões lógicas:

```
RE0 =   RE6 & RE7   (AND)
RE1 =   RE6 | RE7   (OR)
RE2 =   RE6 ^ RE7   (XOR)
RE3 = ~(RE6 | RE7)  (NOR)
```

Parte II

1. Escreva e teste um programa, em *assembly*, que, para além das necessárias inicializações, implemente os seguintes contadores (atualizados com uma frequência de 4 Hz, i.e. a cada 0.25 s):
   - contador binário crescente de 4 bits;
   - contador Johnson de 4 bits (sequência: 0000, 0001, 0011, 0111, 1111, 1110, 1100, 1000, 0000, 0001, ...)

   (utilize os *system calls* de manipulação do *Core Timer* para gerar a frequência de 4Hz)
2. Faça as alterações ao programa que achar convenientes de modo a que seja possível escolher, através do bit 6 do porto E, qual dos dois contadores deverá estar em execução. O bit 7 do porto E deverá controlar a direção da contagem (0 ascendente, 1 descendente). Sempre que haja alteração no valor lido do bit 6 do porto E, para além da comutação de comportamento, o contador deve ser inicializado a 0000.

Parte III

A função seguinte gera um atraso programável, em múltiplos de 1ms, cujo valor é passado como argumento:

```
// Geracao de um atraso programável:
//  - valor mínimo: 1 ms (para n_intervals = 1)
//  - valor máximo: (2^32-1) * 1ms (para n_intervals = 0xFFFFFFFF)
//    (aproximadamente 4294967 s, i.e., 49.7 dias :) )
//
void delay(unsigned int n_intervals)
{
   volatile unsigned int i;

   for(; n_intervals != 0; n_intervals--)
      for(i = CALIBRATION_VALUE; i != 0; i--)
         ;
}
```

A função deverá portanto demorar a executar um tempo de **`n_intervals * 1ms`**, ou seja, o ciclo interno deverá demorar a executar **`1ms`**. Para que o atraso gerado seja correto há pois necessidade de determinar o valor da constante **`CALIBRATION_VALUE`**. Para isso, poderá ser obtida uma estimativa bastante aproximada usando os *system calls* de manipulação do *Core Timer*. Repare que o *Core Timer* é incrementado com uma frequência de relógio de 20 MHz (i.e., metade da frequência de relógio do CPU) pelo que 1ms corresponde a 20000 ciclos desse sinal.

Nota: A palavra-chave **`volatile`** dá a indicação ao compilador que a variável pode ser alterada de forma não explicitada na zona de código onde está a ser usada (i.e., noutra zona de código, como por exemplo numa rotina de serviço à interrupção). Com esta palavra-chave força-se o compilador a, sempre que o valor da variável seja necessário, efetuar o acesso à posição de memória onde essa variável reside, em vez de usar uma eventual cópia residente num registo interno do CPU. Na função anterior, sem o **`volatile`** um compilador inteligente eliminaria os dois ciclos **`for`** visto que não têm efeitos secundários (apenas gastam tempo, que é coisa que os compiladores tentam minimizar).

1. Traduza para *assembly* do MIPS a função **`delay()`**, utilizando para a constante **`CALIBRATION_VALUE`** o valor 6000. Escreva também o programa principal que invoque, em ciclo infinito, a função **`delay()`**:

```
void main(void)
{
   while(1)
   {
      resetCoreTimer();
      delay(1);
      printInt(readCoreTimer(), 10 + (10 << 16));
      printStr("\r\n");
   }
}
```

2. Execute o programa e tome nota do valor impresso.

3. Repita a execução do programa com um novo valor para a constante, por exemplo 12000. Tome nota do valor produzido pelo programa e, com os dois pares de valores (constante, valor lido do *CoreTimer*) determine a equação da recta que une os dois pontos (declive e ordenada na origem). A partir dessa equação extrapole o valor correto da constante **`CALIBRATION_VALUE`**, para obter um valor lido do *CoreTimer* de 20000. Com essa constante, o atraso gerado pela função **`delay()`** será muito próximo de 1 ms.

4. Outra possibilidade para calibrar a função de geração do atraso consiste em usar um porto de saída de 1 bit (por exemplo o RE4) para gerar uma onda quadrada, e o osciloscópio do laboratório para medir o tempo a 1 e o tempo a 0 desse sinal.

   Traduza para *assembly* o trecho de código seguinte, usando a função **`delay()`** com a constante que determinou no ponto anterior. Compile, transfira para a placa DETPIC32 e execute esse código. Com o osciloscópio no porto RE4 meça o tempo a 1 e o tempo a 0 do sinal. Proceda de forma semelhante à descrita no ponto 3 e extrapole o valor da constante de maneira a obter exactamente 1ms em cada um desses tempos.

```
void main(void)
{
   int v = 0;

   TRISE4 = 0;    // Configura o porto RE4 como saída
   while(1)
   {
      LATE4 = v;  // Escreve v no bit 4 do porto E
      delay(1);   // Atraso de 1ms
      v ^= 1;     // complementa o bit 0 de v
   }
}
```

Nota. No que respeita à configuração e ao acesso aos portos de I/O, o programa anterior não está escrito de forma compatível com o compilador de C que será utilizado nas aulas práticas de AC2. A forma correcta de o fazer será descrita no trabalho prático n.º 4.

Elementos de apoio

- PIC32 Family Reference Manual, Section 12 – I/O Ports.
- PIC32MX5XX/6XX/7XX, Family Data Sheet, Pág. 96 a 101.
- *Slides* das aulas teóricas.

Adenda ao trabalho prático N.º 3

1. Cada *led* do *display* de 7 segmentos tem de ter em série uma resistência; a não colocação desta resistência tem como consequência a destruição dos *displays* e/ou da saída do microcontrolador à qual o led estiver ligado. A corrente consumida por cada led pode ser calculada como: I = (3.3 – 1.5) / R; para uma resistência de 270 ohm, a corrente no *led* é, aproximadamente, 6.7 mA.
2. Não deve utilizar qualquer fonte de tensão externa. Deve-se ter, no entanto, em atenção que o consumo máximo admissível do conjunto não deverá exceder 100 mA.
3. A inserção/remoção da placa DETPIC32 na placa branca deve ser realizada com muito cuidado de forma a evitar o empeno e consequente quebra dos seus pinos. A inserção deve ser feita pressionando lentamente a placa em ambas as extremidades A remoção deve ser evitada, mas sempre que houver necessidade de o fazer, deverá ser efetuada de forma a puxar simultaneamente as duas extremidades da placa DETPIC32.
4. Sugestão de organização do espaço nas placas brancas:

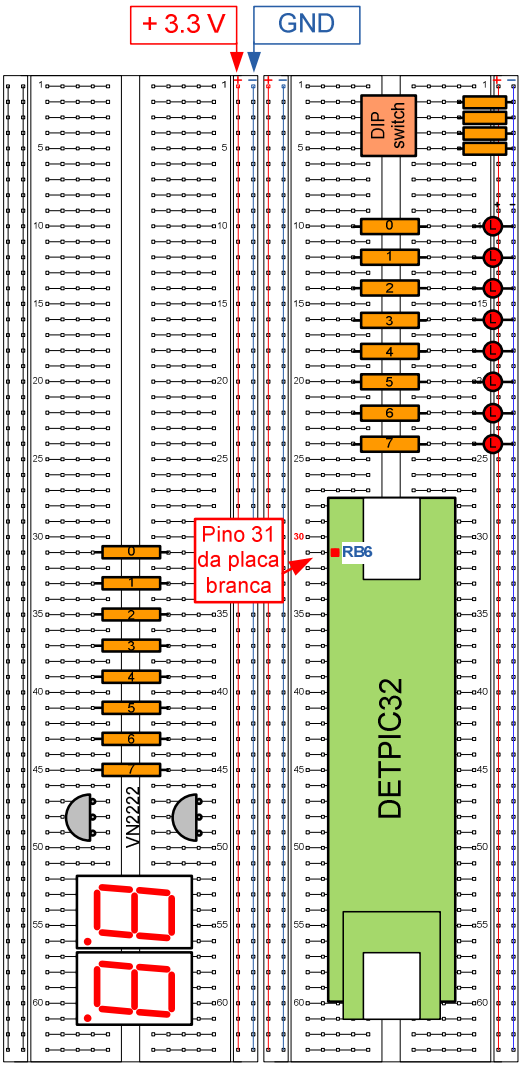


Figura 3. A placa DETPIC32 deverá ser montada de tal forma que a ficha de ligação USB fique posicionada na extremidade da placa branca. Por outro lado, a ligação à placa DETPIC32 estará menos sujeita a erros se o pino 1 (correspondente ao porto RB6) ficar posicionado no pino 31 da placa branca. Deste modo, para se obter o número do pino da placa DETPIC32 basta subtrair 30 ao número da pista correspondente da placa branca.

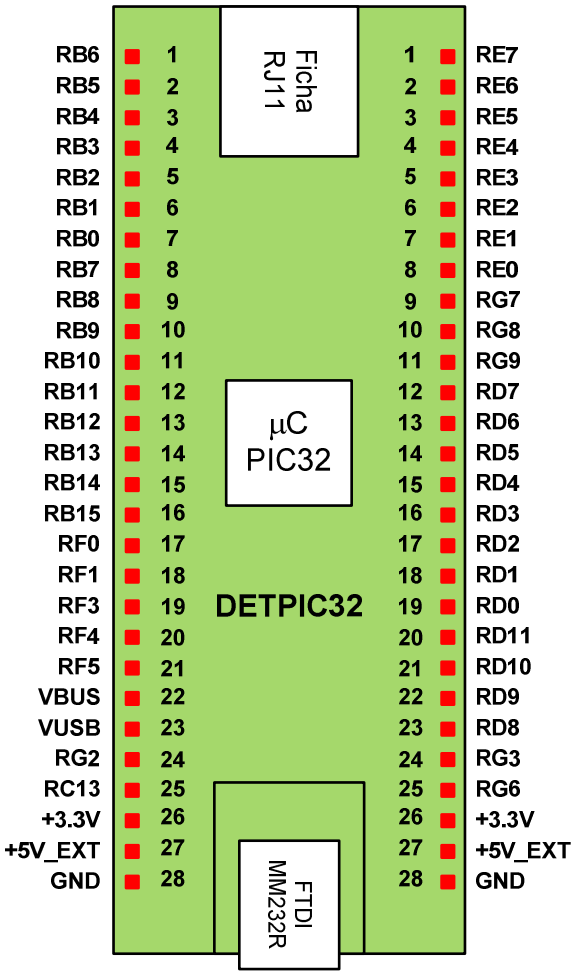


Figura 4. Disposição dos pinos de ligação à placa branca no DETPIC32.

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").

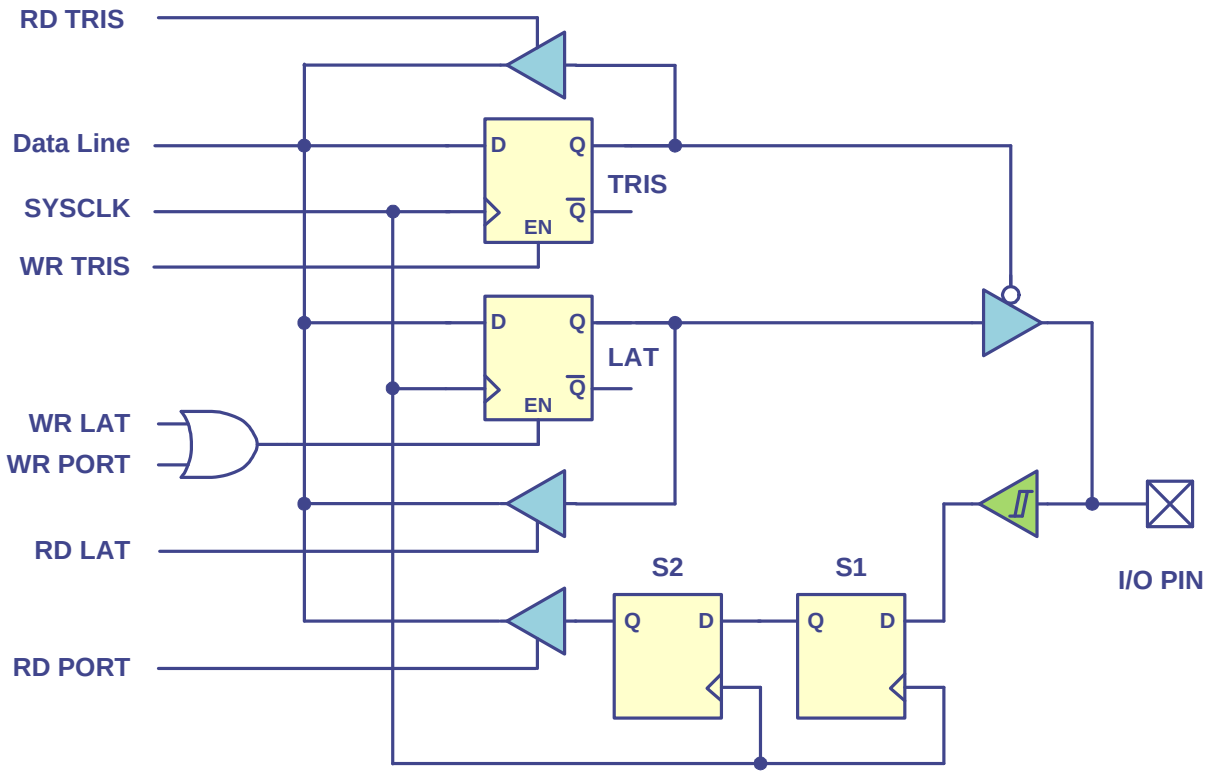


Figura 5. Diagrama de blocos simplificado de um porto de I/O no PIC32.

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).

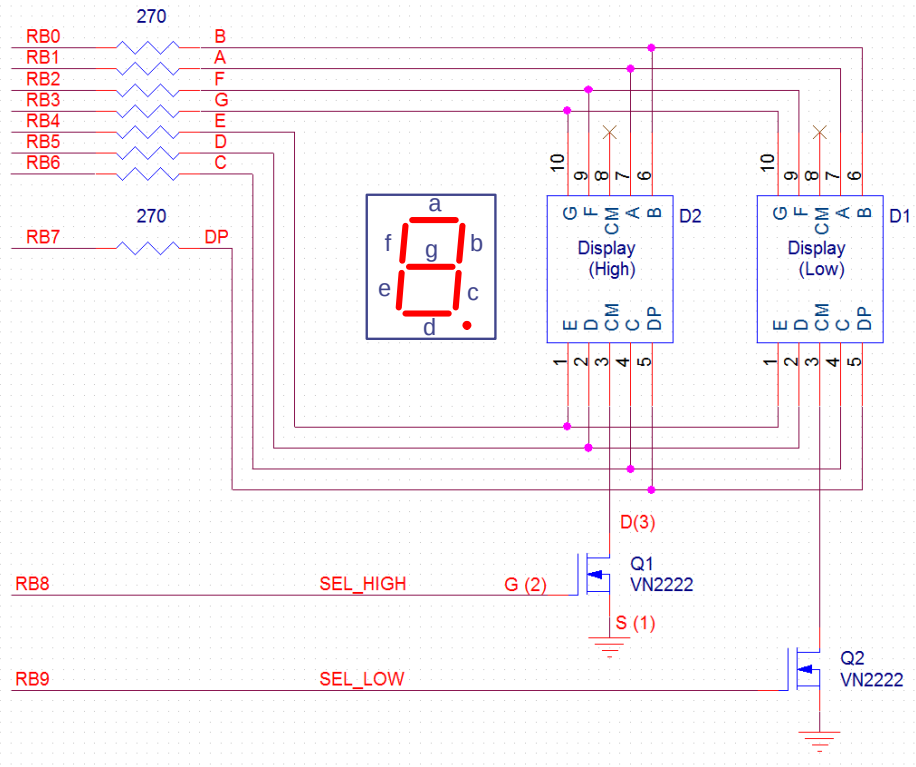


Figura 6. Ligação de dois *displays* de 7 segmentos ao porto B do PIC32.

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.

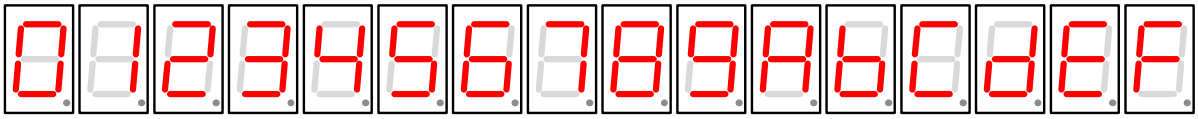


Figura 7. Representação dos dígitos de 0 a F no *display* de 7 segmentos.

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):

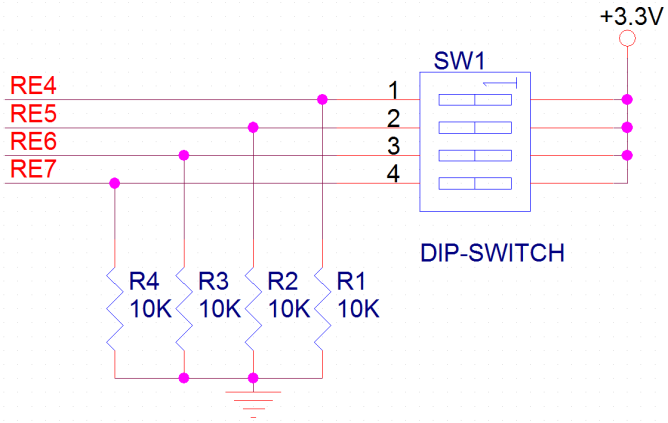


Figura 8. *Dip-switch* de 4 posições ligado a 4 bits do porto E.

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).

Trabalho prático N.º 5

Objetivos

- Familiarização com o modo de funcionamento de um periférico com capacidade de produzir informação.
- Utilização da técnica de *polling* para detetar a ocorrência de um evento e efetuar o consequente processamento.
- Efectuar a conversão analógica/digital de um sinal de entrada e mostrar o resultado no sistema de visualização implementado anteriormente.

Introdução

Um conversor analógico-digital (A/D) é um dispositivo electrónico que efetua a conversão de uma quantidade contínua (uma tensão) numa representação digital com **n** bits (uma quantidade numérica com **n** bits). O número de bits que o conversor usa para a representação numérica designa-se por resolução e representa o logaritmo na base 2 do número de níveis em que o conversor divide a gama de tensão de entrada (quanto maior for a resolução, melhor será a exactidão da conversão). Por exemplo, um conversor A/D de 10 bits divide a gama de tensão de entrada em 1024 níveis (2^{10}), fazendo corresponder à tensão mínima o valor 0x000 e à tensão máxima admissível o valor 0x3FF (1023). Se os valores mínimo e máximo dessas tensões forem 0V e 3.3V, respetivamente, as correspondentes representações digitais serão 0x000 e 0x3FF. A Figura 9 mostra um exemplo de codificação de uma gama de tensão Vmax com 3 bits, isto é, com 8 níveis.

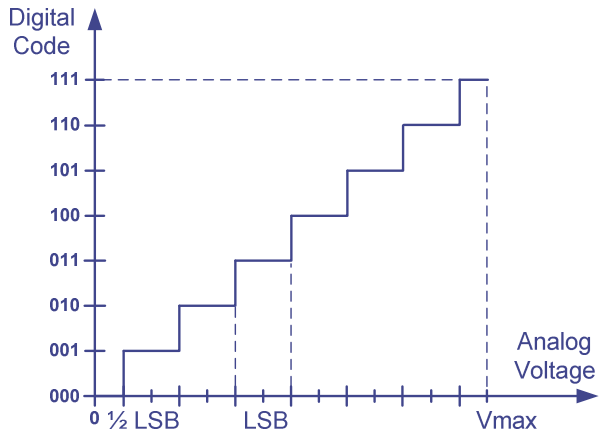


Figura 9. Conversão de uma gama de tensão 0 - Vmax com 3 bits (8 níveis).

O incremento na tensão de entrada a que corresponde uma alteração no bit menos significativo da codificação designa-se por LSB e é dado por LSB = VR / (2^N-1), em que VR é a gama de tensão de entrada e N o número de bits usado para a codificação. No exemplo da Figura 9, se Vmax for igual a 7V então LSB será LSB=7 / (2^3-1) = 1 V. Para um conversor de 10 bits com uma gama de tensão de entrada de 3.3V, o valor do LSB é dado por LSB = 3.3V / (2^{10} -1), aproximadamente 3.2 mV.

O PIC32 disponibiliza um módulo de conversão analógico-digital, com um modelo de programação que permite múltiplas possibilidades de configuração. No essencial, o módulo A/D é constituído por um conversor analógico-digital de 10 bits (um conversor de aproximações sucessivas), um *multiplexer* analógico de 16 entradas e uma zona de memória onde o conversor coloca o(s) resultado(s) da conversão, tal como esquematizado no diagrama de blocos da Figura 10. A zona de memória (designada por *buffer*) é constituída por 16 registos de

32 bits, referenciados pelos nomes **ADC1BUF0**, **ADC1BUF1**, ..., **ADC1BUFF**, que podem ser acedidos nos endereços **0xBF809070**, **0xBF809080**, ..., **0xBF809160**.

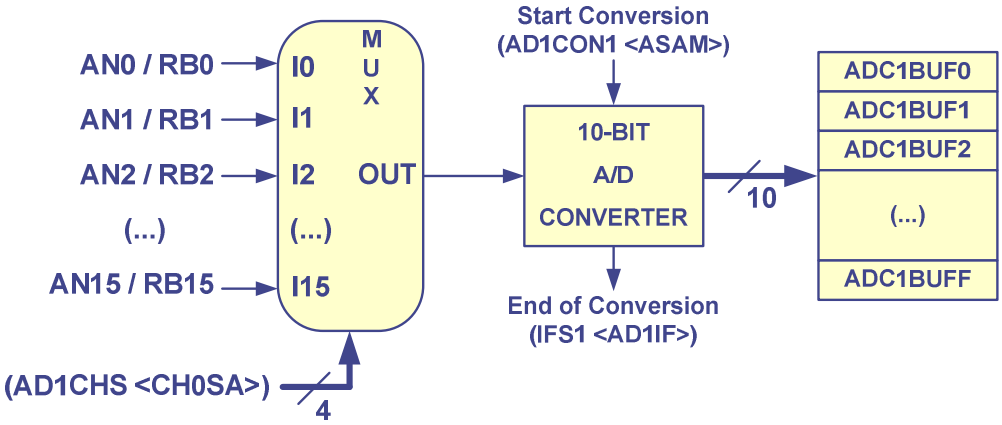


Figura 10. Diagrama de blocos simplificado do módulo A/D do PIC32.

O objetivo deste trabalho prático não é o estudo aprofundado do módulo A/D, pelo que vamos usar apenas um dos vários modos de funcionamento possíveis. Algumas das indicações que serão dadas ao longo deste texto são válidas apenas para o modo de funcionamento escolhido (para informações mais detalhadas deve ser consultado o manual do fabricante disponível no site de AC2). Para esse modo de funcionamento, o processo de conversão envolve:

1) selecionar a entrada analógica a converter, isto é, escolher o canal de entrada (registo **AD1CHS**, bits **CH0SA**);
2) dar ordem de início de conversão ao conversor A/D (registo **AD1CON1**, bit **ASAM**);
3) esperar que o sinal que indica fim de conversão fique ativo (registo **IFS1**, bit **AD1IF**);
4) ler o resultado da conversão em um ou mais registos **ADC1BUFx**.

Configuração das entradas analógicas

As 16 entradas analógicas do PIC 32 são fisicamente coincidentes com os 16 bits do porto B. Qualquer um destes 16 pinos pode ser configurado como entrada analógica ou como porto digital, sendo que na placa DETPIC32 o porto B, por omissão, está configurado como porto digital. A configuração completa de um dado bit **x** do porto B como entrada analógica envolve sempre dois passos: 1) desligar a componente digital de saída do porto, isto é, fazer **TRISBx=1** e 2) configurar o porto como entrada analógica. A configuração como entrada analógica é efetuada através do registo **AD1PCFG**. Por exemplo, para a configuração do bit 4 do porto B como entrada analógica (AN4) pode fazer-se:

```
TRISBbits.TRISB4  = 1;  // RB4 digital output disconnected
AD1PCFGbits.PCFG4 = 0;  // RB4 configured as analog input (AN4)
```

Selecção do canal de entrada

O *multiplexer* analógico permite selecionar, em cada instante, qual a entrada analógica que é encaminhada para o conversor A/D. A selecção do canal de entrada é efetuada através dos 4 bits do campo **CH0SA** do registo **AD1CHS**. Por exemplo, para a seleção da entrada AN4 como entrada para o conversor A/D pode fazer-se:

```
AD1CHSbits.CH0SA = 4;   // Selects AN4 as input for the A/D converter
```

Configuração do número de conversões consecutivas do mesmo canal

O módulo A/D permite configurar o número de conversões consecutivas do mesmo canal antes de o conversor gerar o evento de fim de conversão. Essa configuração é efetuada no registo **AD1CON2** nos 4 bits do campo **SMPI**. Os valores possíveis de configuração, em binário, vão, assim, desde **0000** a **1111**, a que correspondem 1 e 16 conversões consecutivas, respectivamente. Por exemplo, para efetuar 4 conversões consecutivas no canal 7 pode fazer-se:

```
AD1CHSbits.CH0SA = 7;   // Selects AN7 as input for the A/D converter
AD1CON2bits.SMPI = 3;   // 4 samples will be converted and stored
                        // in buffer locations ADC1BUF0 to ADC1BUF3
```

O *buffer* de armazenamento referido anteriormente, é preenchido em função do número de conversões que tiver sido previamente configurado, começando sempre em **ADC1BUF0**. No exemplo de cima, o sinal de fim de conversão só é gerado quando o conversor A/D tiver efetuado as 4 conversões.

Início de conversão e deteção de fim de conversão

Na configuração adoptada para estas aulas práticas, o módulo A/D funciona em modo manual. Significa isto que um processo de conversão só começa quando há uma ordem específica de início. Para essa configuração, a ordem de conversão é dada através da instrução:

```
AD1CON1bits.ASAM = 1;     // Start conversion
```

Quando o módulo A/D termina uma sequência de conversão gera um pedido de interrupção (activa o bit **AD1IF** do registo **IFS1)**. Este pedido de interrupção pode ou não ter seguimento, dependendo da configuração do sistema de interrupções. Se não estiverem a ser usadas interrupções, a deteção do evento de fim de conversão terá quer ser feita por *polling*, esperando, em ciclo, que o bit **AD1IF** transite do nível lógico 0 para o nível lógico 1. O ciclo de *polling* pode ser efetuado do seguinte modo:

```
while( IFS1bits.AD1IF == 0 );  // Wait while conversion not done
```

O bit **IFS1** é automaticamente ativado pelo módulo A/D, mas a sua desativação é manual. Assim, terminada a operação de leitura dos valores da sequência de conversão, é sempre necessário fazer o *reset* desse bit (**IFS1bits.AD1IF = 0**).

Configuração completa do módulo A/D

Para além das apresentadas anteriormente, há ainda um conjunto de configurações adicionais que têm que ser efetuadas para que o módulo A/D funcione de acordo com o pretendido. A lista completa de configurações do módulo A/D fica então:

```
AD1CON1bits.SSRC = 7;     // Conversion trigger selection bits: in this
                          //  mode an internal counter ends sampling and
                          //  starts conversion
AD1CON1bits.CLRASAM = 1;  // Stop conversions when the 1st A/D converter
                          //  interrupt is generated. At the same time,
                          //  hardware clears the ASAM bit
AD1CON3bits.SAMC = 16;    // Sample time is 16 TAD (TAD = 100 ns)
AD1CON2bits.SMPI = XX-1;  // Interrupt is generated after XX samples
                          //  replace XX by the desired number of
                          //  consecutive samples
AD1CHSbits.CH0SA = YY;    // replace YY by the desired input
                          //  analog channel (0 to 15)
AD1CON1bits.ON = 1;       // Enable A/D converter
                          //  This must the last command of the A/D
                          //  configuration sequence
```

Trabalho a realizar

1. No circuito da figura seguinte uma resistência variável (designada por potenciómetro) está ligada ao bit AN14 do PIC32 (RB14). Na configuração em que está montada, a resistência variável permite variar, de forma linear, a tensão no seu ponto intermédio entre 0 V e 3.3 V. Monte esse circuito e ligue o ponto intermédio do potenciómetro à entrada RB14 da placa DETPIC32.

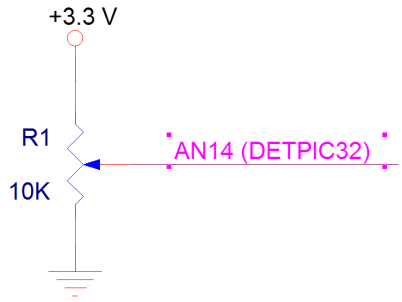


Figura 11. Ligação da resistência variável à placa DETPIC32.

2. Escreva um programa que:
   a) configure o porto RB14 como entrada analógica; configure o módulo A/D de acordo com as indicações dadas anteriormente e de modo a que o número de conversões consecutivas seja 1;
   b) em ciclo infinito: 1) dê ordem de conversão; 2) espere pelo fim de conversão; 3) utilizando o *system call* **`printInt()`**, imprima o resultado da conversão (disponível em **`ADC1BUF0`**) em hexadecimal, formatado com 3 dígitos.

```
int main(void)
{
   // Configure A/D module and RB14 as analog input
   while(1)
   {
      // Start conversion
      // Wait while conversion not done (AD1IF == 0)
      // Read conversion result (ADC1BUF0 value) and print it
      // Reset AD1IF
   }
}
```

   Rode gradualmente o potenciómetro e observe os valores produzidos pelo programa.

3. Meça o tempo de conversão do conversor A/D. Para isso, configure um bit como saída digital (por exemplo RE0) e faça a ativação desse porto durante o tempo em que a conversão está a ser realizada (comente a linha de código correspondente à impressão do valor). Com o osciloscópio meça o tempo de ativação de RE0, a que corresponde, com boa aproximação, o tempo total de conversão, e tome nota desse valor.

```
int main(void)
{
   volatile int aux;
   // Configure A/D module; configure RE0 as digital output
   while(1)
   {
      // Set RE0
      // Start conversion
      // Wait while conversion not done (AD1IF == 0)
      // Reset RE0
      // Read conversion result (ADC1BUF0) to "aux" variable
      // Reset AD1IF
   }
}
```

4. Altere o programa anterior de modo a imprimir as 16 posições do *buffer* do módulo A/D (valores impressos em decimal, formatados com 4 dígitos, separados por 1 espaço). Relembre que os endereços dos 16 registos do *buffer* de armazenamento dos resultados são múltiplos de 16 (**0xBF809070**, **0xBF809080**, ..., **0xBF809160)**. Pode fazer do seguinte modo:

```
int *p = (int *)(&ADC1BUF0);
for( i = 0; i < 16; i++ )
{
   printInt( p[i*4], ... )
   ...
}
```

5. Altere a configuração do módulo A/D de modo a que o conversor efetue 4 conversões consecutivas. Observe o resultado.

6. O valor fornecido pelo conversor é, como já foi referido anteriormente, a representação digital com 10 bits da amplitude da tensão na sua entrada. Sabendo que a tensão máxima à entrada é 3.3V, pode facilmente determinar-se a amplitude da tensão que deu origem a um valor produzido pelo conversor A/D: **V=(VAL_AD*3.3)/1023**. O cálculo de V utilizando apenas inteiros obriga a usar uma representação em vírgula fixa com, por exemplo, uma casa decimal. Para isso, a expressão anterior pode ser re-escrita do seguinte modo: **V=(VAL_AD*33)/1023** ou, com arredondamento, **V=(VAL_AD*33+511)/1023**.

   Retome o programa que escreveu no exercício 5 e acrescente código para calcular a média das 4 amostras retiradas e para determinar a amplitude da tensão e imprimir o seu valor usando *system calls*.

7. Integre no programa anterior o sistema de visualização que desenvolveu no trabalho prático n.º 4. Faça as alterações que permitam a visualização do valor da amplitude da tensão nos *displays* de 7 segmentos. O programa deverá efetuar 4 sequências de conversão A/D por segundo (cada uma delas com 4 amostras) e o sistema de visualização deverá funcionar com uma frequência de refrescamento de 100 Hz (10 ms).

```
void main(void)
{
   // Configure all (digital I/O, analog input, A/D module)
   i = 0;
   while(1)
   {
      // Wait 10 ms using the core timer
      if(i++ == 25)  // 250 ms
      {
         // Convert analog input (4 samples)
         // Calculate buffer average
         // Calculate voltage amplitude
         // Convert voltage amplitude to decimal
         // i = 0;
      }
      // Send voltage value to displays
   }
}
```

Note: Antes de se iniciar uma nova sequência de conversão, o ou os valores resultantes da sequência de conversão anterior têm que ser obrigatoriamente lidos, na sua totalidade, do *buffer* de resultados. Enquanto essa operação de leitura não for feita, o módulo A/D ignora qualquer comando posterior de início de conversão.

Trabalho prático N.º 6

Objetivos

- Familiarização com o modo de funcionamento de um periférico com capacidade de produzir informação.
- Utilização da técnica de interrupção para detetar a ocorrência de um evento e efetuar o consequente processamento.
- Efectuar a conversão analógica/digital de um sinal de entrada e mostrar o resultado no sistema de visualização implementado anteriormente.

Introdução

Como mencionado no trabalho prático anterior, quando o módulo A/D termina uma sequência de conversão gera um pedido de interrupção (ativa o bit **AD1IF** do registo **IFS1)**. Para que este pedido de interrupção tenha seguimento, o sistema de interrupções do microcontrolador terá que estar devidamente configurado, de modo a que, na ocorrência do evento de fim de conversão, a rotina de serviço à interrupção (*Interrupt Service Routine*, ISR) seja executada.

Trabalho a realizar

1. No trabalho prático anterior fizemos a deteção do evento de fim de conversão por *polling*, isto é, num ciclo que espera pela passagem a 1 do bit **AD1IF**. O que se pretende agora é que o atendimento ao evento de fim de conversão seja feito por interrupção e não por *polling*. Para isso, para além das configurações já efetuadas anteriormente, é ainda necessário:
   a) configurar o nível de prioridade das interrupções geradas pelo módulo A/D – registo **IPC6**[6], nos 3 bits **AD1IP** (terá que ser um valor entre 1 e 6; o valor 7, a que corresponde a prioridade máxima, não deve ser usado; para o valor 0 os pedidos de interrupção nunca são aceites);
   b) autorizar as interrupções geradas pelo módulo A/D – registo **IEC1**, no bit **AD1IE**;
   c) ativar globalmente o sistema de interrupções;

   O esqueleto de programa que se apresenta de seguida mostra a estrutura-base do programa para interagir com o módulo A/D por interrupção. Neste primeiro exercício pretende-se, tal como já se fez no exercício 2 do trabalho prático anterior, que o módulo A/D gere a interrupção ao fim de 1 conversão (SMPI=0). A rotina de serviço à interrupção imprime o valor lido do conversor e dá nova ordem de aquisição ao módulo A/D.

```
void main(void)
{
   // Configure all (digital I/O, analog input, A/D module, interrupts)
   ...
   IFS1bits.AD1IF = 0;               // Reset AD1IF flag
   EnableInterrupts();               // Global Interrupt Enable
   // Start A/D conversion
   while(1)
      ;  // do nothing (all activity is done by the ISR)
}
```

[6] A informação relativa a cada fonte de interrupção, nomeadamente o vetor associado e registos de configuração, está condensada na tabela das páginas 122 a 124 do PIC32MX5XX/6XX/7XX, Family Data Sheet (disponível no site de AC2).

```
// Interrupt Handler

void _int_(VECTOR) isr_adc(void)      // Replace VECTOR by the A/D vector
                                      // number - see "PIC32 family data
                                      // sheet" (pages 122-124)
{
   // Print ADC1BUF0 value            // Hexadecimal (3 digits format)
   // Start A/D conversion
   IFS1bits.AD1IF = 0;                // Reset AD1IF flag
}
```

O uso de _int_(VECTOR) indica ao compilador que a função que se segue é uma rotina de serviço a uma interrupção, pelo que o compilador, entre outras coisas, emite o código necessário para salvaguardar todos os registos que são usados por essa função.

2. No exercício 3 do trabalho prático anterior mediu-se o tempo de conversão do conversor A/D. Sabendo esse tempo podemos agora estimar a latência no atendimento a uma interrupção no PIC32 (intervalo de tempo que decorre desde o pedido de interrupção até à execução da primeira instrução "útil" da rotina de serviço à interrupção). Para isso, vamos usar novamente um porto digital configurado como saída (por exemplo o RE0). Desactive o bit RE0 à entrada da rotina de serviço à interrupção e active-o à saída.

```
void _int_(VECTOR) isr_adc(void)
{
   // Reset RE0                       // RE0 = 0
   // Print ADC1BUF0 value            // Hexadecimal (3 digits format)
   // Set RE0                         // RE0 = 1
   // Start A/D conversion
   IFS1bits.AD1IF = 0;                // Reset AD1IF flag
}
```

Execute o programa e, com um osciloscópio, meça o tempo durante o qual o bit RE0 permanece ao nível lógico 1 e tome nota desse valor. Se subtrair a esse tempo o tempo de conversão medido no exercício 3 do trabalho prático anterior, obtém a latência do atendimento a uma interrupção no PIC32. Sabendo que a frequência do CPU é 40 MHz, poderá explicitar o resultado em termos do número de ciclos de relógio.

3. Pretende-se agora estimar o *overhead* global do antendimento a uma interrupção no PIC32. Para isso, e para além da latência, temos ainda de considerar o tempo necessário para o regresso ao programa interrompido, essencialmente constituído pelo tempo necessário para repor o contexto salvaguardado no início da rotina de serviço à interrupção.

   Para medir esse tempo podemos activar um porto de saída (por exemplo o RE0) no fim da rotina de serviço à interrupção (deve ser a última instrução dessa rotina) e desactivar esse mesmo porto no ciclo infinito do programa principal. Meça, com o osciloscópio, o tempo durante o qual o porto RE0 está activo e expresse esse tempo em número de ciclos de relógio. Adicionando esse valor ao obtido no ponto anterior, obtém uma boa estimativa para o *overhead* global no atendimento a uma interrupção no PIC32.

4. Integre no programa anterior o sistema de visualização. Faça as alterações que permitam a visualização do valor da amplitude da tensão nos *displays* de 7 segmentos. O programa deverá efetuar 4 sequências de conversão A/D por segundo (cada uma com 8 amostras consecutivas) e o sistema de visualização deverá funcionar com uma frequência de refrescamento de 100 Hz (10 ms). Utilize, na organização do seu código, o programa-esqueleto que se apresenta de seguida:

```
volatile unsigned char value2display = 0;   // Global variable

void main(void)
{
   // Configure all (digital I/O, analog input, A/D module, interrupts)
   ...
   IFS1bits.AD1IF = 0;               // Reset AD1IF flag
   EnableInterrupts();               // Global Interrupt Enable
   i = 0;
   while(1)
   {
      // Wait 10 ms using the core timer
      if(i++ == 25)  // 250 ms
      {
         // Start A/D conversion
         // i = 0;
      }
      // Send "value2display" variable to displays
   }
}

void _int_(VECTOR) isr_adc(void)
{
   // Calculate buffer average (8 samples)
   // Calculate voltage amplitude
   // Convert voltage amplitude to decimal. Assign it to "value2display"
   IFS1bits.AD1IF = 0;          // Reset AD1IF flag
}
```

A palavra-chave **`volatile`** dá a indicação ao compilador que a variável pode ser alterada de forma não explicitada na zona de código onde está a ser usada (i.e., noutra zona de código, como por exemplo numa rotina de serviço à interrupção). Com esta palavra-chave força-se o compilador a, sempre que o valor da variável seja necessário, efetuar o acesso à posição de memória onde essa variável reside, em vez de usar uma eventual cópia, potencialmente com um valor desatualizado, residente num registo interno do CPU.

Elementos de apoio

- Slides das aulas teóricas.
- PIC32 Family Reference Manual, Section 17 – A/D Module.
- PIC32 Family Reference Manual, Section 08 – Interrupts.
- PIC32MX5XX/6XX/7XX, Family Data Sheet, Pág. 122 a 124.

Trabalho prático N.º 7

Objetivos

- Programação e utilização de *timers*.
- Utilização das técnicas de *polling* e de interrupção para detetar a ocorrência de um evento e efetuar o consequente processamento.
- Geração de sinais PWM.

Introdução

Timers são dispositivos periféricos de grande utilidade em aplicações baseadas em microcontroladores permitindo, por exemplo, a geração de eventos de interrupção periódicos ou a geração de sinais PWM (*Pulse Width Modulation*) com *duty-cycle* variável. O seu funcionamento baseia-se na contagem de ciclos de relógio de um sinal com frequência conhecida. O PIC32 disponibiliza 5 *timers*, T1 a T5, que podem ser usados para a geração periódica de eventos de interrupção ou como base de tempo para a geração de sinais PWM. Esta última funcionalidade está reservada aos timers T2 e T3 e é implementada num módulo à parte, designado pelo fabricante por *Output Compare Module*.

No PIC32MX795F512H (versão que equipa a placa DETPIC32), os *timers* T2 a T5 são do tipo B e o T1 é do tipo A. A principal diferença entre o *timer* de tipo A e os restantes reside no módulo *prescaler* que apenas permite, no de tipo A, a divisão por 1, 8, 64 ou 256. Os *timers* do tipo B podem ser agrupados 2 a dois implementando, desse modo, um *timer* de 32 bits. A Figura 12 apresenta o diagrama de blocos simplificado de um *timer* tipo B do PIC32.

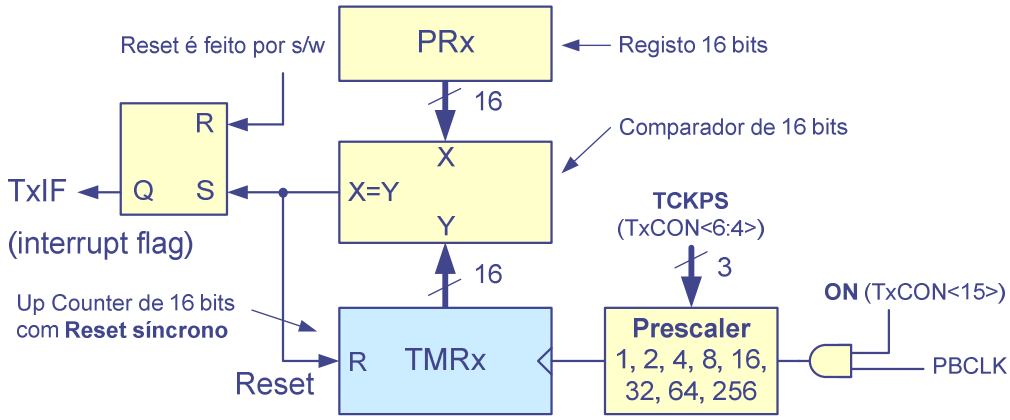


Figura 12. Diagrama de blocos simplificado de um timer tipo B.

Nesta visão simplificada, a fonte de relógio para os *timers* é apenas o *Peripheral Bus Clock* (PBCLK) que, na placa DETPIC32, está configurado para ter uma frequência igual a metade da frequência do sistema, isto é, f_{PBCLK} = 20 MHz (igual a **FREQ/2** ou a **PBCLK** em C).

Cálculo das constantes para geração de um evento periódico

O módulo de pré-divisão (*prescaler*) efetua uma divisão da frequência f_{PBCLK} por um fator configurável nos 3 bits **TCKPS** do registo **TxCON**, para os *timers* T2 a T5, ou nos 2 bits **TCKPS** do registo **T1CON**, para o *timer* T1. Conhecida a frequência do sinal à saída do *prescaler*, pode determinar-se a frequência do sinal gerado pelo *timer*, do seguinte modo:

$$f_{OUT} = f_{IN} / (PRx + 1)$$

em que f_{IN} é a frequência do sinal à saída do *prescaler* e **PRx** é o valor da constante de 16 bits armazenada num dos registos **PR1** a **PR5** (*timers* T1 a T5).

Exemplo: determinar o valor da constante **PR2** de modo a que o *timer* T2 gere interrupções a uma frequência de 10 Hz.

Se o *prescaler* for configurado com o valor 1, então f_{IN} = 20 MHz e **PR2** fica:

PR2 = (20 x 10^6 / 10) – 1 ≅ 2x10^6 (em C, PR2=PBCLK/10-1;)

Ora, uma vez que o registo **PR2** é de 16 bits, o valor máximo da constante de divisão é 65535 (2^{16}-1), pelo que a solução anterior é impossível. Será então necessário configurar o módulo *prescaler* de modo a baixar a frequência do sinal à entrada do contador do *timer*, de modo a tornar possível a divisão com uma constante de 16 bits. Se, por exemplo, se usar um fator de divisão de 32, então f_{IN} = 20 MHz / 32 = 625 KHz. Refazendo o cálculo para o valor de PR2 obtém-se:

PR2 = (625 x 10^3 / 10) – 1 = 62499 (em C, PR2=PBCLK/32/10-1;)

valor que já é possível armazenar num registo de 16 bits.

A obtenção de um evento com a mesma frequência no *timer* T1 obrigava à utilização de um fator de divisão de 64, uma vez que o valor 32 não está disponível nesse *timer*.

Configuração do *timer*

A programação dos *timers* envolve a configuração de alguns bits do registo **TxCON**, bem como a configuração da constante de divisão **PRx**. A sequência para a configuração do *timer* T2 com os parâmetros do exemplo anterior é:

```
T2CONbits.TCKPS = 5; // 1:32 prescaler (i.e. fin = 625 KHz)
PR2 = 62499;         // Fout = 20MHz / (32 * (62499 + 1)) = 10 Hz
TMR2 = 0;            // Reset timer T2 count register
T2CONbits.TON = 1;   // Enable timer T2 (must be the last command of the
                     //  timer configuration sequence)
```

Configuração do *timer* para gerar interrupções

Se se pretender que o *timer* gere interrupções é necessário, para além da configuração-base apresentada no ponto anterior, configurar o sistema de interrupções na parte respeitante ao *timer* ou *timers* que estão a ser usados, nomeadamente, prioridade (registo IPCx), *enable* das interrupções geradas pelo *timer* pretendido (registo IECx) e *reset* inicial do bit TxIF (registo IFSx)[7]. Para o *timer* T2, a sequência de instruções que activam o sistema de interrupções fica então:

```
IFS0bits.T2IF = 0;   // Reset timer T2 interrupt flag
IPC2bits.T2IP = 2;   // Interrupt priority (must be in range [1..6])
IEC0bits.T2IE = 1;   // Enable timer T2 interrupts
```

Geração de um sinal PWM

PWM (*Pulse Width Modulation*, ou modulação por largura de impulso) é uma técnica usada em múltiplas aplicações, desde o controlo de potência a fornecer a uma carga à geração de efeitos de áudio ou à modulação digital em sistemas de telecomunicações. Esta técnica utiliza sinais rectangulares, como o apresentado na Figura 13, em que, mantendo o período T, se pode alterar dinamicamente a duração a 1, t_{ON}, do sinal.

[7] Para saber quais os registos que deve configurar para um *timer* em particular deve consultar o manual do fabricante "PIC32, Family Reference Manual Section 14-Timers", ou o "PIC32MX5XX/6XX/7XX, Family Data Sheet", Pág. 122 a 124 (ambos disponíveis no site da disciplina).

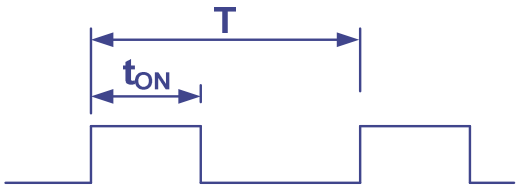


Figura 13. Exemplo de sinal rectangular com um período T e um tempo a 1 t_{ON}.

O *duty-cycle* de um sinal PWM é definido pela relação entre o tempo durante o qual o sinal está no nível lógico 1 (num período) e o período desse sinal, e expressa-se em percentagem:

Duty-cycle = (t_{ON} / T) * 100 [%]

No PIC32 a geração de sinais PWM é efetuada usando os *timers* T2 e T3 e o *Output Compare Module* (OC). A Figura 14 apresenta o diagrama de blocos do sistema de geração de sinais PWM, onde se evidencia a interligação entre o módulo correspondente aos *timers* T2 e T3 e o módulo OC.

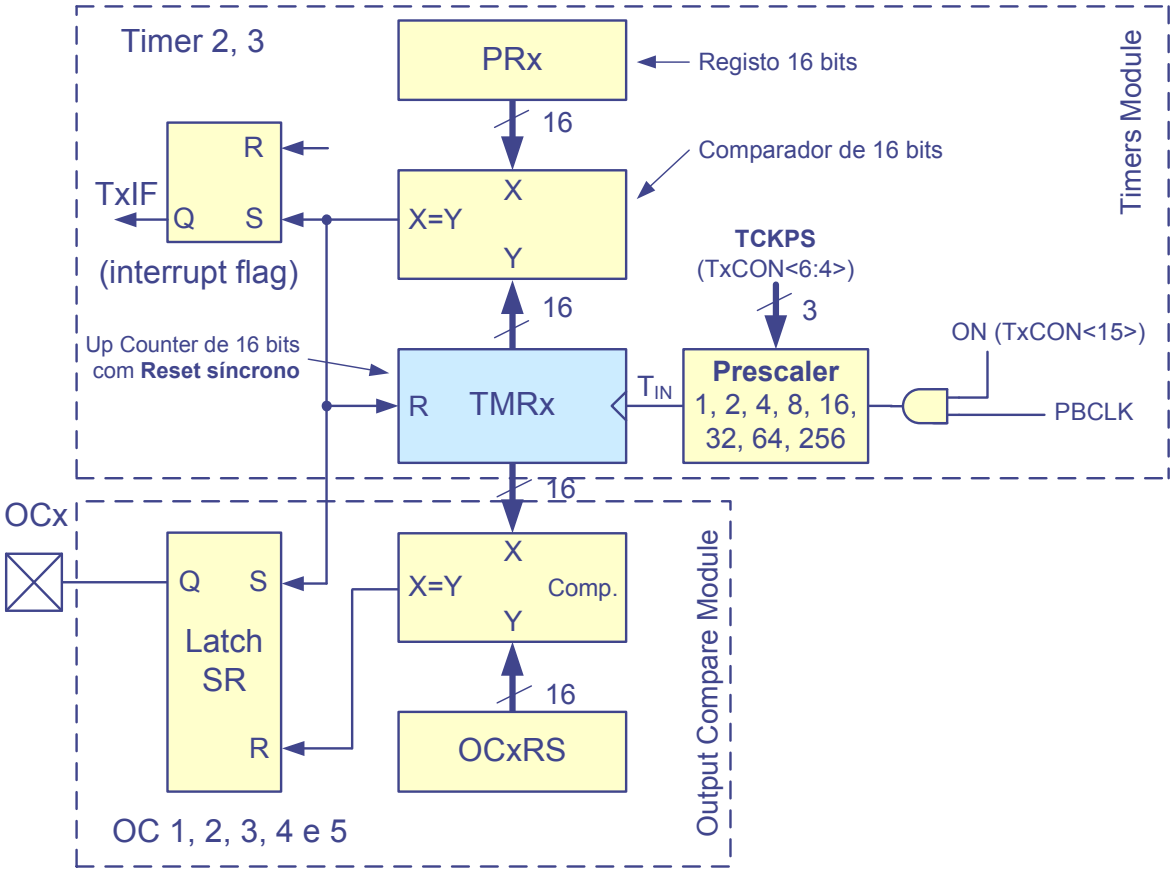


Figura 14. Diagrama de blocos do sistema de geração de sinais PWM.

Nesta forma de organização do sistema de geração de sinais PWM, um dos *timers* T2 ou T3 funciona como base de tempo, isto é, define o período T do sinal, enquanto que o módulo OC permite configurar, através do registo OCxRS, a duração a 1 desse sinal, isto é, o tempo t_{ON}.

Exemplo: determinar as constantes relevantes para a geração, na saída OC1, de um sinal com uma frequência de 10 HZ e um *duty-cycle* de 20%, usando como base de tempo o *timer* T2.

O valor de PR2, que determina a frequência do sinal de saída, foi já calculado anteriormente (62499). Temos então que calcular o valor da constante a colocar no registo **OC1RS**:

$t_{ON} = 0.2 * T_{PWM} = 0.2 * (1 / 10) = 20$ ms

$f_{IN} = 625$ KHz, $T_{IN} = 1 / 625000 = 1.6$ µs

Então **OC1RS** deverá ser configurado com:

$OC1RS = 20*10^{-3}/1.6*10^{-6} = 12500$

Alternativamente, poderemos simplesmente multiplicar o valor de (**PRx + 1**) pelo valor do *duty-cycle* pretendido. Neste caso ficaria:

OC1RS = ((PR2 + 1) * duty_cycle)/100 = (62499 + 1) * 0.2 = 12500

Conhecendo os valores da frequência do sinal de saída (PWM) e do sinal à entrada do contador, pode calcular-se a resolução com que o sinal PWM pode ser gerado:

$Resolução = \log_2(T_{PWM} / T_{IN}) = \log_2(f_{IN} / f_{OUT})$

Para as frequências do exemplo anterior a resolução é então: $\log_2(625000 / 10) = 15.9$ bits

A sequência completa de programação para obter o sinal de 10 Hz e *duty-cycle* de 20% na saída OC1 fica então:

```
T2CONbits.TCKPS = 5; // 1:32 prescaler (i.e fin = 625 KHz)
PR2 = 62499;         // Fout = 20MHz / (32 * (62499 + 1)) = 10 Hz
TMR2 = 0;            // Reset timer T2 count register
T2CONbits.TON = 1;   // Enable timer T2 (must be the last command of the
                     //  timer configuration sequence)
OC1CONbits.OCM = 6;  // PWM mode on OCx; fault pin disabled
OC1CONbits.OCTSEL =0;// Use timer T2 as the time base for PWM generation
OC1RS = 12500;       // Ton constant
OC1CONbits.ON = 1;   // Enable OC1 module
```

O valor do registo **OC1RS** pode ser modificado, sem qualquer problema, em qualquer altura, sem necessidade de se alterar qualquer um dos outros registos. Isso permite a alteração dinâmica do *duty-cycle* do sinal gerado, em função das necessidades.

As saídas OC1 a OC5 estão fisicamente multiplexadas com os bits RD0 a RD4 do porto D (pela mesma ordem). A ativação do *Output Compare Module* OCx configura automaticamente o porto correspondente como saída, não sendo necessária qualquer configuração adicional (esta configuração sobrepõe-se à efetuada através do registo TRISD).

Trabalho a realizar

Parte I

1. Calcule as constantes relevantes e configure o *timer* T3, de modo a gerar eventos com uma frequência de 2 Hz. Em ciclo infinito, faça *polling* do bit de fim de contagem **T3IF** e envie para o ecrã o caracter **'.'** sempre que esse bit fique ativo:

```
void main(void)
{
   // Configure Timer T3 (2 Hz, interrupts disabled)
   while(1)
   {
      // Wait until T3IF == 1
      // Reset T3IF
      putChar('.');
   }
}
```

2. Substitua o atendimento por *polling* por atendimento por interrupção, configurando o *timer* T3 para gerar interrupções à frequência de 2 Hz.

```
void main(void)
{
   // Configure Timer T3, interrupts
   EnableInterrupts();
   while(1);
}

void _int_(VECTOR) isr_T3(void) // Replace VECTOR by the timer T3
                                //  vector number
{
   putChar('.');
   // Reset T3 interrupt flag
}
```

3. Altere o programa anterior de modo a que o *system call* putChar() seja evocado com uma frequência de 1 Hz.
4. Retome agora o exercício 4 do trabalho prático n.º 6. Nesse exercício implementou-se um sistema para adquirir 4 sequências de conversão A/D por segundo (cada uma delas com 8 amostras) e visualizar o valor da tensão, calculado a partir da média da sequência de conversão, nos *displays* de 7 segmentos. Por seu lado, o sistema de visualização funcionava com uma frequência de refrescamento de 100 Hz (10 ms). Ainda nesse exercício, os tempos relevantes (10 ms e 0.25 s) eram controlados por *polling*, usando o *Core Timer*.

   Pretende-se agora a utilização de *timers* com atendimento por interrupção para controlar o funcionamento do sistema. Assim, comece por calcular as constantes relevantes para que o *timer* T1 (tipo A) gere eventos de interrupção a cada 250 ms (4 Hz) e o *timer* T3 (tipo B) gere eventos de interrupção a cada 10 ms (100 Hz).

   Faça as correspondentes alterações ao programa que escreveu no último exercício dos "guiões 5 e 6", de modo a utilizar os *timers* T1 e T3 com atendimento por interrupção.

```
volatile unsigned char value2display = 0;   // Global variable

void main(void)
{
   configureAll(); // Function to configure all (digital I/O, analog
                   //  input, A/D module, timer T1, timer T3, interrupts)
   // Reset AD1IF, T1IF and T3IF flags
   EnableInterrupts();    // Global Interrupt Enable
   while(1);
}

void _int_(VECTOR_ADC) isr_adc(void)
{
   // Calculate buffer average (8 samples)
   // Calculate voltage amplitude
   // Convert voltage amplitude to decimal. Assign it to "value2display"
   IFS1bits.AD1IF = 0;               // Reset AD1IF flag
}

void _int_(VECTOR_TIMER1) isr_T1(void)
{
   // Start A/D conversion
   // Reset T1IF flag
}
```

```
void _int_(VECTOR_TIMER3) isr_T3(void)
{
   // Send "value2display" global variable to displays
   // Reset T3IF flag
}
```

5. Implemente a função *freeze* que "congela" nos *displays* o último valor de tensão convertido pelo módulo A/D. Para isso configure os portos RE5 e RE4 como entrada e faça as alterações ao código que permitam parar a aquisição quando o valor lido desses dois portos tiver a combinação binária 01 (RE5=0; RE4=1). Sugestão: controle o bit de *enable*/*disable* das interrupções do *timer* T1.

Parte II

1. Escreva um programa que gere na saída OC1 (pino RD0 da placa DETPIC32) um sinal com uma frequência de 100 Hz e um *duty-cycle* de 25%, utilizando como base de tempo o *timer* T3. Observe o sinal com o osciloscópio e verifique se os tempos do sinal (período e tempo a 1, t_{ON}) estão de acordo com o programado.
2. Escreva uma função que permita (para a frequência de 100Hz) configurar o módulo OC1 para gerar qualquer valor de PWM entre 0 e 100, passado como argumento.

```
void setPWM(unsigned int dutyCycle)
{
   // duty_cycle must be in the range [0, 100]
   OC1RS = ...;   // Evaluate OC1RS as a function of "dutyCycle"
}
```

3. Teste a função anterior com outros valores de *duty-cycle*, por exemplo, 10%, 65% e 80%. Observe, para os diferentes valores de *duty-cycle*, que o brilho do LED D1 depende do valor do *duty-cycle* do sinal de PWM gerado. Para todos os valores de *duty-cycle* meça, com o osciloscópio, o tempo t_{ON} do sinal.
4. Pretende-se agora integrar o controlo do brilho do LED no programa que escreveu no ponto 5 da parte 1. Para isso, os bits RE5 e RE4 vão ser usados para escolher o modo de funcionamento do sistema:

```
00 – funciona como voltímetro (o LED deve ficar OFF)
01 – congela o valor atual da tensão (o LED deve ficar ON com o brilho
     no máximo)
10 - controlo do brilho do LED (dependente dos bits RE7 e RE6)
11 – para uso futuro
```

5. O brilho do LED depende, como já observámos anteriormente, do *duty-cycle* do sinal que o controla. Assim, para controlar o brilho do LED, gere um sinal (com a mesma frequência de 100 Hz) cujo *duty-cycle* dependa da combinação binária presente nos bits RE7 e RE6, do seguinte modo:

```
00 – Duty Cycle = 03%
01 – Duty Cycle = 15%
10 – Duty Cycle = 40%
11 – Duty Cycle = 90%
```

No modo de controlo do brilho do LED o valor do PWM deve ser visualizado nos *displays* de 7 segmentos.

Na página seguinte apresenta-se o esqueleto da função **`main()`** que pode usar para implementar esta funcionalidade.

```
volatile unsigned int value2display;

void main(void)
{
   const static unsigned char pwmValues[]={3, 15, 40, 90};

   configureAll();
   EnableInterrupts();  // Global Interrupt Enable
   while(1)
   {
      // Read RE5, RE4 to the variable "portVal"
      switch(portVal)
      {
         case 0:  // Measure input voltage
            // Enable T1 interrupts
            setPWM(0);
            break;

         case 1:  // Freeze
            // Disable T1 interrupts
            setPWM(100);
            break;

         case 2:  // LED brigthness control
            // Disable T1 interrupts
            // Read RE7, RE6 (duty-cycle value) to the variable "dc"
            setPWM(pwmValues[dc]);
            // Copy duty-cycle value to global variable "value2display"
            break;

         default:
            break;
      }
   }
}
```

Note: a variável **"value2display"** serve para colocar o valor a mostrar nos *displays*. Essa variável é atualizada, de forma alternativa, na rotina de serviço à interrupção do módulo A/D ou no programa principal quando o modo de controlo de brilho do LED está ativo.

Elementos de apoio

- Slides das aulas teóricas.
- PIC32 Family Reference Manual, Section 08 – Interrupts.
- PIC32 Family Reference Manual, Section 14 – Timers.
- PIC32 Family Reference Manual, Section 17 – A/D Module.
- PIC32MX5XX/6XX/7XX, Family Data Sheet, Pág. 122 a 124.

Trabalho prático N.º 8

Objetivos

- Compreender os mecanismos básicos que envolvem a comunicação série assíncrona.
- Implementar funções básicas de comunicação série através de uma UART, usando as técnicas de *polling* e de interrupção.

Introdução

A UART (acrónimo que significa *Universal Asynchronous Receiver Transmitter*) é um canal de comunicação série assíncrona *full-duplex* e é um dos módulos disponíveis no PIC32 que permitem comunicação série. Implementa o protocolo RS232 o que permite, por exemplo, a ligação a um PC (directamente através de uma porta RS232 ou indirectamente usando uma ligação física USB). O PIC32, na versão usada na placa DETPIC32, disponibiliza 6 UARTs, das quais a UART1 assegura, através de um conversor USB - RS232, a comunicação com o PC.

A UART é constituída, essencialmente, por 3 módulos fundamentais: um módulo de transmissão (TX), um módulo de receção (RX) e um gerador do sinal de relógio para a transmissão e receção, vulgarmente designado por gerador de *baudrate* (BRG – *BaudRate Generator*).

Módulo de transmissão

Na Figura 15 apresenta-se o diagrama de blocos simplificado do módulo de transmissão que inclui, para possibilitar uma visão mais completa do sistema, a ligação ao módulo BRG.

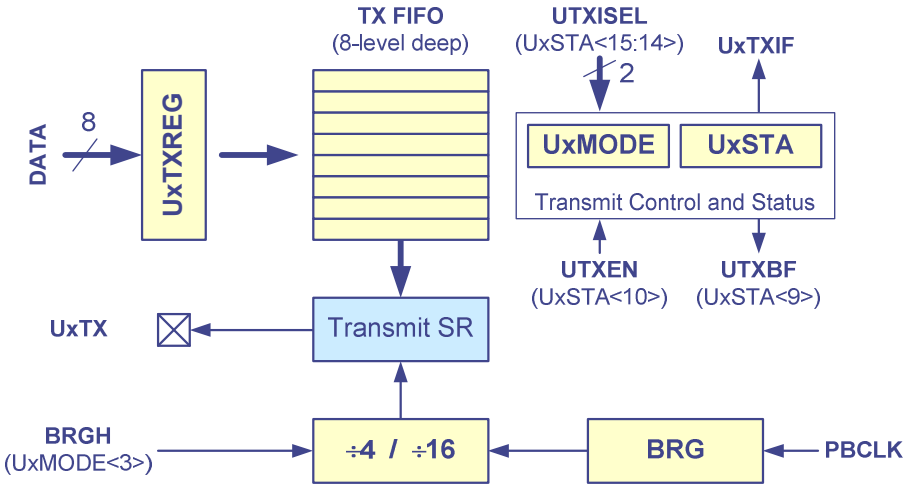


Figura 15. Diagrama de blocos simplificado do módulo de transmissão da UART (perspectiva do programador).

O bloco-base do módulo de transmissão é um *shift register* (*Transmit SR*) que faz a conversão paralelo-série, enviando sucessivamente para a linha série os caracteres[8] armazenados no *buffer* de transmissão. Este *buffer* é um FIFO (*First In First Out*) de 8 posições, gerido de forma automática pelo *hardware* e a que o programador acede, indirectamente, através do registo **UxTXREG**[9]. Assim, o envio de informação é feito escrevendo, sucessivamente, os caracteres a transmitir no registo **UxTXREG**, que funciona como a cauda do FIFO ("TX FIFO"). Os caracteres armazenados no FIFO são copiados sucessivamente para o *shift-register*.

[8] A UART permite a comunicação com palavras de 8 e 9 bits. Nestas aulas consideramos apenas a comunicação com palavras de 8 bits, a que chamamos, caracteres.

[9] A letra **x** minúscula que aparece no nome de todos os registos deve ser substituída pelo número da UART pretendida (entre 1 e 6). Por exemplo, para a UART1 o registo de transmissão é U1TXREG.

O relógio que determina a taxa de transmissão é obtido através de uma divisão por 4 ou por 16 (em função do bit de configuração **BRGH** do registo **UxMODE**) do relógio gerado pelo módulo BRG.

A ativação deste módulo é efectuada ativando o bit **UTXEN** do registo **UxSTA** – após *reset* ou *power-on* todas as 6 UART estão desligadas e os módulos de transmissão e recepção de cada uma delas não estão activos.

Módulo de receção

A Figura 16 apresenta o diagrama de blocos simplificado do módulo de receção onde se inclui também o módulo BRG. Como se pode observar, a estrutura do módulo de receção é semelhante ao módulo de transmissão, tendo como elemento central um *shift-register* que faz, neste caso, a conversão série-paralelo. Os sucessivos caracteres recebidos da linha série são colocados no *buffer* de receção que é, tal como no módulo de transmissão, um FIFO de 8 posições ("RX FIFO"). A gestão do "RX FIFO" é, igualmente, realizada pelo *hardware* de forma automática e transparente para o programador que interage com esta estrutura de dados indirectamente através do registo **UxRXREG**.

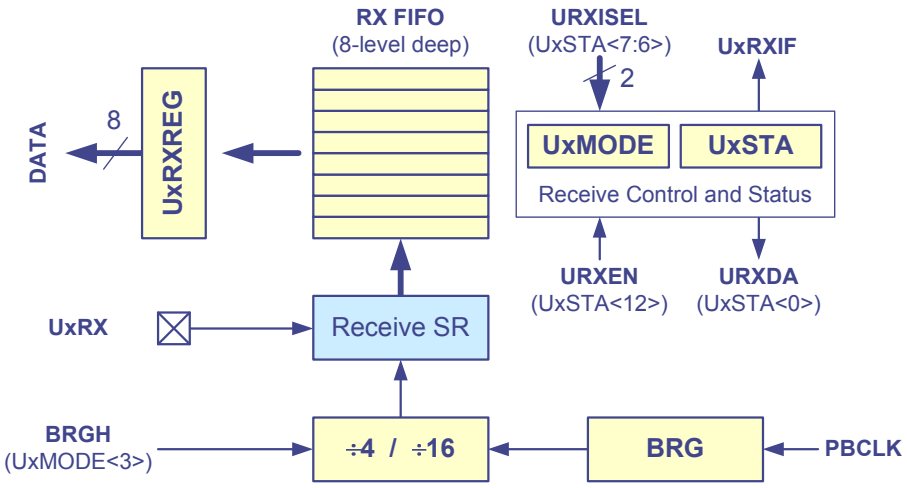


Figura 16. Diagrama de blocos simplificado do módulo de receção da UART (perspectiva do programador).

O relógio que determina a taxa de receção é obtido através de uma divisão por 4 ou por 16 (em função do bit de configuração **BRGH** do registo **UxMODE**) do relógio gerado pelo módulo BRG.

O bit **URXEN** do registo **UxSTA** ativa este módulo que, tal como o de transmissão, fica inativo após *reset* ou *power-on*.

Gerador de *baudrate*

O gerador de *baudrate* apresenta uma estrutura semelhante aos *timers* de utilização geral já apresentados e utilizados no trabalho prático anterior, tal como se pode observar na Figura 17.

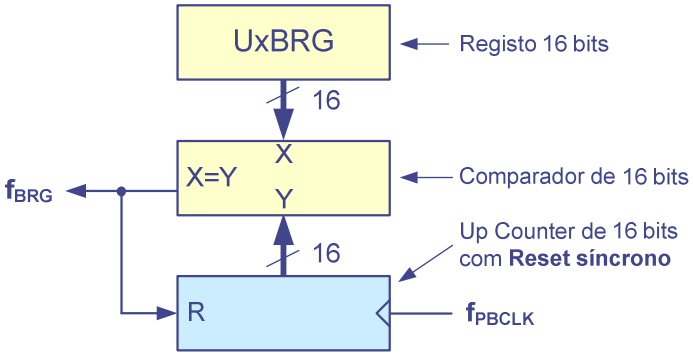


Figura 17. Diagrama de blocos do módulo gerador de *baudrate*.

O sinal de relógio à entrada deste módulo é o *Peripheral Bus Clock* que tem, na placa DETPIC32, uma frequência de 20 MHz.

A frequência à saída deste módulo é, então,

$$f_{BRG} = f_{PBCLK} / (UxBRG + 1)$$

em que **UxBRG** representa a constante armazenada no registo com o mesmo nome. O sinal à saída deste módulo é posteriormente dividido por 4 ou por 16, em função da configuração do bit **BRGH** do registo **UxMODE**, representando a frequência do sinal obtido a taxa de transmissão/recepção (*baudrate*) da UARTx.

No caso em que **BRGH** (**UxMODE<3>**) é configurado com o valor 1, o factor de divisão referido anteriormente é 4, pelo que a taxa de transmissão/receção é dada por:

$$baudrate = f_{PBCLK} / (4 * (UxBRG + 1))$$

No caso em que **BRGH** (**UxMODE<3>**) é configurado com o valor 0, o factor de divisão é 16, sendo então a taxa de transmissão/receção dada por:

$$baudrate = f_{PBCLK} / (16 * (UxBRG + 1))$$

ou

$$UxBRG = (f_{PBCLK} / (16 * baudrate)) - 1$$

Ou ainda, e de modo a evitar o erro de truncatura que se terá ao realizar as operações com inteiros:

$$UxBRG = ((f_{PBCLK} + 8 * baudrate) / (16 * baudrate)) - 1$$

O módulo BRG é comum aos módulos de transmissão e receção, pelo que a taxa de transmissão é igual à taxa de receção.

Configuração da UART

A configuração completa da UART, sem interrupções, é efetuada nos seguintes passos :

1) Configurar o gerador de *baudrate* de acordo com a taxa de transmissão/receção pretendida (registo **UxBRG** e bit **BRGH** do registo **UxMODE**).
2) Configurar a dimensão da palavra a transmitir (número de *data bits*), tipo de paridade (bits **PDSEL<1:0>** do registo **UxMODE**) e o número de *stop bits* (bit **STSEL** do registo **UxMODE**).
3) Ativar os módulos de transmissão e receção (bits **UTXEN** e **URXEN** do registo **UxSTA**).
4) Ativar a **UART** (bit **ON** do registo **UxMODE**).

A ativação de uma dada UARTx e dos correspondentes módulos de transmissão e receção configura automaticamente o porto de transmissão (**UxTX**) como saída e o porto de receção (**UxRX**) como entrada, sobrepondo-se esta configuração à efetuada através do(s) registo(s) **TRISx**.

Procedimento de transmissão (*polling*)

O procedimento de transmissão envolve sempre a verificação da existência de espaço no respetivo FIFO. O bit **UTXBF** (*Transmit Buffer Full*) do registo **UxSTA** é um bit de *status* que, quando ativo, indica que o FIFO de transmissão está cheio. É, assim, necessário esperar que o bit **UTXBF** fique inativo para colocar nova informação no FIFO. Assim, uma função para transmitir 1 caracter deverá ser estruturada do seguinte modo:

```
void putc(char byte2send)
{
   // wait while UTXBF == 1
   // Copy byte2send to the UxTXREG register
}
```

Procedimento de receção (*polling*)

No caso da receção, o bit **URXDA** (*Receive Data Available*) do registo **UxSTA** indica, quando ativo, que está disponível para ser lido pelo menos 1 caracter no FIFO de receção. O procedimento genérico de leitura envolve, assim, o *polling* desse bit, esperando que ele fique ativo para então ler o caracter recebido. A estrutura de uma função bloqueante de leitura de 1 caracter poderá então ser a seguinte:

```
char getc(void)
{
   // Wait while URXDA == 0
   // Return U1RXREG
}
```

Na situação em que o FIFO de receção está cheio e um novo caracter foi lido da linha série, ocorre um erro de *overrun*, sinalizado no bit **OERR** do registo **UxSTA**. Nesta situação, o último caracter lido da linha série é descartado e, enquanto o bit **OERR** estiver activo, a UART não recebe mais nenhum caracter. Assim, a função de receção de um caracter deve incluir o teste da *flag* **OERR** de modo a efetuar o respectivo *reset* no caso de ocorrência de erro de *overrun* (o *reset* dessa *flag* elimina toda a informação existente no *buffer* de receção). A função anterior poderá então ser re-escrita do seguinte modo:

```
char getc(void)
{
   // If OERR == 1 then reset OERR
   // Wait while URXDA == 0
   // Return U1RXREG
}
```

Configuração da UART com interrupções

O modo como as interrupções são geradas é configurável através dos bits **UTXISEL<1:0>** e **URXISEL<1:0>** do registo **UxSTA**. Configurando esses bits com a combinação binária "**00**", são geradas interrupções nas seguintes situações:

- Transmissão: uma interrupção é gerada quando o FIFO de transmissão tem, pelo menos, uma posição livre; a rotina de serviço à interrupção pode, assim, colocar no registo **UxTXREG** um novo caracter para ser transmitido;
- Receção: uma interrupção é gerada quando o FIFO de receção tem, pelo menos, um novo caracter para ser lido; a rotina de serviço à interrupção pode efetuar a respetiva leitura do registo **UxRXREG**.

Para além destas duas configurações específicas, é ainda necessário configurar, como para todas as outras fontes de interrupção, os bits de *enable* das interrupções e os bits que definem a prioridade. Assim, para a ativação da interrupção de receção é necessário configurar o bit **UxRXIE** (*receive interrupt enable*) e para a interrupção de transmissão o bit **UxTXIE** (*transmit interrupt enable*). Para a definição da prioridade devem ser configurados os 3 bits **UxIP** (a configuração de prioridade é comum a todas as fontes de interrupção de uma UART).

Cada UART pode ainda gerar uma interrupção quando é detetada uma situação de erro na receção de um caracter. Os erros detetados são de três tipos: erro de paridade, erro de *framing*

e erro de *overrun*. Se se pretender fazer a deteção destes erros por interrupção, então é também necessário ativar essa fonte de interrupção, isto é, ativar o bit **UxEIE**.

Finalmente, é importante referir que a cada UART está atribuído um único vetor para as 3 possíveis fontes de interrupção. Essa situação obriga a que a identificação da fonte de interrupção tenha que ser feita por *software* na rotina de serviço à interrupção associada ao vetor da UART que está a ser usada. A identificação é, assim, efetuada através do teste dos *interrupt flag bits* de cada uma das três fontes possíveis de interrupção, isto é, **UxRXIF**, **UxTXIF** e **UxEIF**.

Trabalho a realizar

1. Configure a UART1 (sem interrupções), com os seguintes parâmetros de comunicação: 115200 bps, sem paridade, 8 *data bits*, 1 *stop bit* (consulte os registos **UxMODE** e **UxSTA** no manual da UART). No cálculo da constante de configuração do gerador de *baudrate* considere um fator de divisão do relógio de 16.

```
int main(void)
{
   // Configure UART1:
   // 1 - Configure BaudRate Generator
   // 2 – Configure number of data bits, parity and number of stop bits
   //     (see U1MODE register)
   // 3 – Enable the trasmitter and receiver modules (see register U1STA)
   // 4 – Enable UART1 (see register U1MODE)
}
```

2. Acrescente ao código que escreveu no exercício anterior uma função para o envio de um caracter para a porta série. No programa principal envie, usando essa função, o caracter '**+**' a uma frequência de 1 Hz.

```
int main(void)
{
   // Configure UART1 (115200, N, 8, 1)
   while(1)
   {
      putc('+');
      // wait 1 s
   }
}

void putc(char byte2send)
{
   // wait while UTXBF == 1
   // Copy byte2send to the UxTXREG register
}
```

3. Escreva e teste uma função para enviar para a porta série uma *string* (*array* de caracteres terminado com o caracter 0). Utilize a função **putc()** para enviar cada um dos caracteres da *string*.

```
void main(void)
{
   // Configure UART1 (115200, N, 8, 1)
   while(1)
   {
      puts("String de teste\n");
      // wait 1 s
   }
}
```

```
void puts(char *str)
{
   ... // use putc() function to send each charater ('\0' should not
       //  be sent)
}
```

4. Generalize o código de configuração da UART1 que escreveu no ponto 1, implementando uma função que aceite como argumentos o *baudrate*, o tipo de paridade e o número de *stop bits* (o número de *data bits* deverá ser fixo e igual a 8). Para a configuração do *baudrate* utilize um fator de divisão do relógio de 16.

   Valores possíveis para os argumentos de entrada da função:

   - Baudrate: 600 a 115200
   - Paridade: 'N', 'E', 'O' (sem paridade, paridade par (*even*), paridade ímpar (*odd*))
   - Stop bits: 1 ou 2

   Valores a considerar por omissão, isto é sempre que os argumentos de entrada não estejam nas gamas definidas anteriormente:

   - Baudrate: 115200
   - Paridade: 'N'
   - Stop bits: 1

```
void configUart(unsigned int baud, char parity, unsigned int stopbits)
{
   // Configure BaudRate Generator
   // Configure number of data bits (8), parity and number of stop bits
   // Enable the trasmitter and receiver modules
   // Enable UART1
}
```

5. Teste a função anterior com o programa que escreveu no ponto 3, com diferentes valores de entrada. Exemplo: **`600,'N',1`**; **`1200,'O',2`**; **`9600,'E',1`**; **`19200,'N',2`**; **`115200,'E',1`**. No PC execute o programa **`pterm`** com os mesmos parâmetros com que configurou a UART1 (por exemplo: **`pterm 1200,O,8,2`**).
6. Neste exercício pretende-se avaliar a forma de funcionamento da UART na transmissão. Para isso vamos medir o tempo que a UART demora a transmitir *strings* com diferente dimensão, partindo sempre de uma situação de repouso, isto é, em que garantidamente a UART não tem nenhuma informação pendente para ser transmitida. Para garantir essa condição de início vamos usar o bit **`TRMT`** do registo **`UxSTA`** (**`UxSTA<8>`**) que, quando a 1, indica que o TX FIFO e o *transmit shift register* estão ambos vazios.

```
int main(void)
{
   configUart(115200,'N',1);   //default "pterm" parameters (8 data bits)
   // config RE0 as output
   while(1)
   {
      // Wait until TRMT == 1
      // Set RE0
      puts("12345");
      // Reset RE0
   }
}
```

   Meça, com o osciloscópio, os tempos a 1 e a 0 do sinal no porto RE0 e anote os valores obtidos. Repita o teste, sequencialmente, com as seguintes *strings*: "123456789", "123456789A", "123456789AB", anotando todos os valores. Compare os valores obtidos e tire conclusões. Determine o tempo necessário para transmitir cada caracter.

7. Repita o exercício anterior configurando a UART com *baudrates* de 57600 e 19200 bps.

8. Retire, do programa anterior, a condição de espera e envie a *string* inicial i.e. "12345". Meça novamente o tempo a 1 do sinal no porto RE0, compare esse valor com o que obteve anteriormente para a mesma *string* e tire conclusões.

```
...
while(1)
{
   // Set RE0
   puts("12345");
   // Reset RE0
}
...
```

9. Acrescente agora a função para ler um caracter da linha série e teste-a. Para isso, configure a UART1 com os parâmetros por omissão do **`pterm`** e, em ciclo infinito, re-envie para a linha série todos os caracteres recebidos (procedimento geralmente referido como "eco dos caracteres"). Altere também a função **`getc()`** para detetar erros na receção de caracteres (paridade, *framing* e overrun).

```
int main(void)
{
   configUart(115200,'N',1);    // default "pterm" parameters
   while(1)
   {
      putc( getc() );
   }
}

char getc(void)
{
   // If OERR == 1 then reset OERR
   // Wait while URXDA == 0
   // If FERR or PERR then read UxRXREG and return 0
   // Return U1RXREG
}
```

10. Configure a UART para gerar uma interrupção na receção de um caracter. Altere o programa anterior para fazer o eco do caracter recebido na respetiva rotina de serviço.

```
int main(void)
{
   configUart(115200,'N',1);    // default "pterm" parameters
                                //  with RX interrupt enabled
   EnableInterrupts();
   while(1);
}

void _int_(VECTOR_UART1) isr_uart1(void)
{
   putc(U1RXREG);
   // Clear UART1 rx interrupt flag
}
```

11. Retome o programa que escreveu no último exercício do trabalho prático n.º 7 e inclua a função de configuração da UART1 e de receção por interrupção, que implementou no exercício anterior. Faça ainda as seguintes alterações:

    a) Envie o valor da tensão medida (i.e. o valor que é enviado para os *displays*) para o PC a um ritmo de 1 Hz. Para isso implemente um contador módulo 100 na rotina de serviço à interrupção do Timer T3 (que é evocada, recorde-se, a cada 10 ms) e envie para a porta série os dois dígitos da variável "**`value2display`**", codificados em ASCII.

b) Quando for lida da porta série a tecla "**L**" envie o valor mínimo e o valor máximo da tensão que o sistema mediu desde o seu arranque. Para isso declare duas variáveis globais, "**voltMin**" e "**voltMax**", inicialize-as adequadamente e atualize-as na rotina de serviço à interrupção do módulo A/D, isto é, na rotina que calcula um novo valor de tensão.

```
void _int_(VECTOR_ADC) isr_adc(void)
{
   (...)
   // Convert voltage amplitude to decimal. Assign it to "value2display"
   // Update variables "voltMin" and "voltMax"
   IFS1bits.AD1IF = 0;               // Reset AD1IF flag
}

void _int_(VECTOR_TIMER3) isr_T3(void)
{
   static int counter = 0;

   // Send "value2display" global variable to displays
   if(++counter == 100)
   {
      counter = 0;
      // send voltage to the serial port UART1
   }
   // Clear T3 interrupt flag
}

void _int_(VECTOR_UART1) isr_uart1(void)
{
   if(U1RXREG == 'L')
   {
      // Send "voltMin" and "voltMax" to the serial port UART1
   }
   // Clear UART1 rx interrupt flag
}
```

12. Pretende-se agora detetar, por interrupção, a ocorrência de erros de comunicação na receção (paridade, *framing* e *overrun*). Altere adequadamente a função de configuração da UART1 (veja a introdução para mais detalhes) e acrescente na rotina de serviço à interrupção o tratamento da interrupção de erro.

```
void _int_(VECTOR_UART1) isr_uart1(void)
{
   // If U1EIF set then
   //    if overrun error then clear OERR flag
   //    else read U1RXREG to a dummy variable
   //    clear UART1 error interrupt flag
   //
   // If U1RXIF set then
   //    if(U1RXREG == 'L')
   //       (...)
   // Clear UART1 rx interrupt flag
}
```

Elementos de apoio

- Slides das aulas teóricas.
- PIC32 Family Reference Manual, Section 08 – Interrupts.
- PIC32 Family Reference Manual, Section 21 – UART.
- PIC32MX5XX/6XX/7XX, Family Data Sheet, Pág. 122 a 124.

Trabalho prático N.º 9

Objetivos

- Compreender e implementar a estrutura básica de um *device-driver.*

Introdução

O objetivo deste trabalho prático é a implementação da estrutura básica de um *device-driver* para a UART do PIC32. Um *device-driver* é um programa que permite a outras aplicações comunicar com um dispositivo *hardware*. O *device-driver* lida com todos os detalhes de implementação, configuração e de funcionamento do dispositivo, e fornece uma interface que permite, a uma aplicação de alto nível, a interação com esse dispositivo de forma independente das suas particularidades - se a interface se mantiver inalterada, a eventual mudança do dispositivo *hardware* não acarreta qualquer mudança no programa de alto nível. Uma outra característica de um *device-driver* reside na utilização de interrupções como modelo de transferência de informação.

No caso de um *device-driver* para a UART, a implementação baseia-se no desacoplamento da transferência de dados entre a aplicação e a UART, sendo a ligação efetuada através de FIFOs. Isto significa que a aplicação interage exclusivamente com os FIFOS através de funções de leitura e escrita. Por seu lado, a transferência de informação entre os FIFOs e a UART é efetuada por interrupção e é da exclusiva responsabilidade do *device-driver*. Os FIFOS podem ser implementados como *buffers* circulares, sendo esta a designação que é dada no restante deste texto.

Transmissão

A Figura 18 representa, esquematicamente, a estrutura da componente de transmissão do *device-driver*. A função **`comDrv_putc()`** escreve a informação a enviar para a UART no *buffer* circular de transmissão (*append*) e mantém atualizadas duas variáveis de gestão desse *buffer*: a variável **`tail`** que indica a posição de escrita do próximo caracter e a variável **`count`** que contém o número de caracteres do *buffer* ainda não enviados para a UART. Estas duas variáveis são incrementadas por cada novo caracter escrito no *buffer*, sendo a variável **`tail`** incrementada em módulo **n** (0, 1, 2, ..., n - 1, 0, 1, ...), em que **n** é a dimensão do *buffer.*

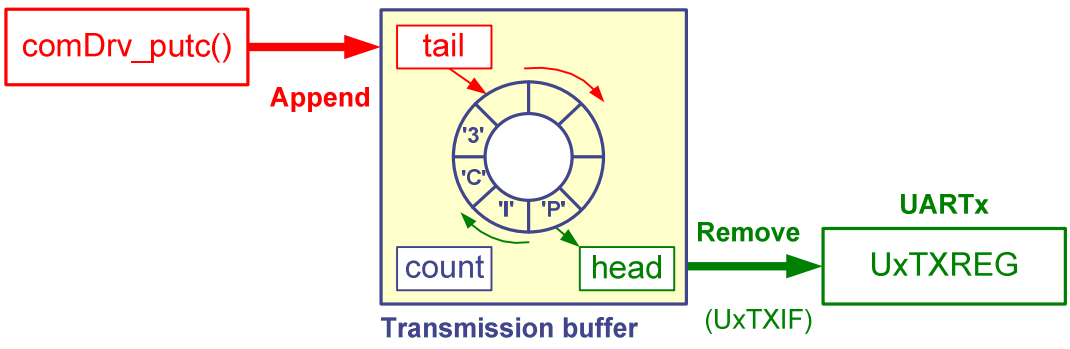


Figura 18. Representação esquemática da componente de transmissão do *device-driver* para a UART.

A informação colocada no *buffer* circular de transmissão é posteriormente enviada, por interrupção, para a UART (*remove*). Assim, a rotina de serviço à interrupção lê o caracter residente na posição **head** do *buffer*, e escreve-o no registo de transmissão da UART. Adicionalmente, por cada caracter transferido para a UART, a rotina de serviço atualiza as variáveis **head** e **`count`**: a variável **`count`** é decrementada e a variável **head** é incrementada em módulo **n** (em que **n** é a dimensão do *buffer*).

Receção

A Figura 19 representa, esquematicamente, a estrutura da componente de receção do *device-driver*. Sempre que a UART recebe um novo caracter gera uma interrupção e, na respetiva rotina de serviço, esse caracter é copiado para o *buffer* de receção para a posição referenciada pela variável **`tail`** (*append*). A rotina de serviço à interrupção mantém atualizadas as variáveis **`tail`** e **`count`** de modo idêntico ao já explicado para o *buffer* de transmissão. Os caracteres presentes no *buffer* de receção são lidos pela aplicação (*remove*) através da função **`comDrv_getc()`** que, de cada vez que lê um caracter, atualiza também as variáveis **`head`** e **`count`**.

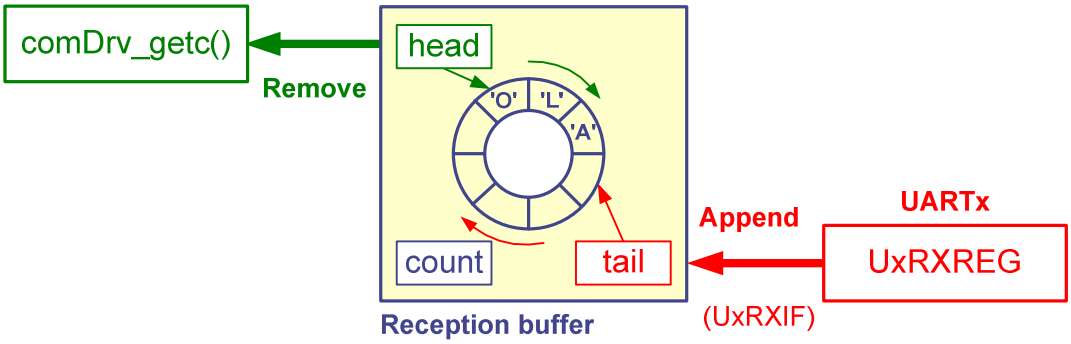


Figura 19. Representação esquemática da componente de receção do *device-driver* para a UART.

Implementação dos *buffers circulares*

Os *buffers* circulares de transmissão e receção têm uma estrutura semelhante. Para a implementação de cada um destes *buffers* é necessário ter uma área de armazenamento (um *array* de caracteres) e um conjunto de variáveis auxiliares de gestão: o índice do *array* onde se pode escrever o próximo caracter (**`tail`**); o índice do *array* de onde se pode ler um caracter (**`head`**); um contador que mantém atualizado o número efetivo de caracteres do *buffer* (**`count`**).

Para a definição da estrutura de um *buffer* circular podemos utilizar, em linguagem C, uma estrutura:

```
typedef struct
{
   unsigned char data[BUF_SIZE];
   unsigned int head;
   unsigned int tail;
   unsigned int count;
} circularBuffer;
```

A partir desta estrutura podem ser instanciados os dois *buffers* circulares do *device driver*:

```
volatile circularBuffer txb;    // Transmission buffer
volatile circularBuffer rxb;    // Reception buffer
```

Uma vez que as variáveis associadas a estes dois *buffers* circulares podem ser alteradas pelas rotinas de serviço à interrupção, tem que se acrescentar na instanciação das estruturas a palavra-chave **`volatile`**. Desse modo, em operações de leitura de qualquer uma dessas variáveis fora da rotina de serviço à interrupção, força-se o compilador a gerar código que efetue o acesso à posição de memória onde a variável reside, em vez de usar uma cópia temporária residente num registo interno do CPU.

A dimensão do *array* de caracteres (**`BUF_SIZE`**) deverá ser ajustada em função das necessidades previsíveis de tráfego. Por uma questão de simplicidade no processamento

associado aos índices de gestão do *array* deve-se, no entanto, estabelecer como dimensão um valor que seja uma potência de 2 (2, 4, 8, 16, ...). A aplicação de uma máscara com o valor da dimensão do *array* menos 1 (**`BUF_SIZE - 1`**, i.e., 1, 3, 7, 15, ...), após uma operação de incremento, garante a rotação do valor do índice, sem qualquer teste adicional.

```
#define BUF_SIZE     32
#define INDEX_MASK   (BUF_SIZE – 1)
```

Receção – rotina de serviço à interrupção

Quando a UARTx gera uma interrupção na receção de um caracter, o caracter recebido deve ser transferido para a posição **`tail`** do *buffer* de receção, o valor do contador de caracteres, **`count`**, deve ser incrementado e o índice de escrita, **`tail`**, deve ser incrementado em módulo **n**. No caso em que o *buffer* fica cheio (situação que ocorre quando os caracteres recebidos e colocados no *buffer* não foram lidos pela aplicação) uma solução para não perder a informação mais recente consiste em descartar o caracter mais antigo. Para isso basta incrementar o índice de leitura **head** (sem incrementar o valor do contador **`count`**).

Receção – leitura do *buffer*

A função **`comDrv_getc()`**, tal como esquematizado na Figura 19, lê um caracter do *buffer* de receção, da posição referenciada pela variável **head**, e atualiza as variáveis **`count`** e **head**. O facto de as variáveis **`count`** e **head** poderem ser alteradas na rotina de serviço à interrupção e na função **`comDrv_getc()`** faz com que elas tenham que ser encaradas como recursos partilhados. Isso tem como consequência que, fora da rotina de serviço à interrupção, uma zona de código que altere essas variáveis seja considerada uma secção crítica. Na secção crítica, não deverá ser permitido o atendimento de interrupções de receção da UARTx. Assim, a interrupção de receção deve ser desativada antes da secção crítica e reativada logo depois.

Transmissão – escrita no *buffer*

A escrita de um caracter no *buffer* de transmissão é efetuada, como indicado na Figura 18, pela função **`comDrv_putc()`**. Esta função copia o caracter a enviar para o *buffer* e atualiza as variáveis **`count`** e **`tail`**. A variável **`count`** pode também ser escrita na rotina de serviço à interrupção, pelo que o código de alteração desta variável fora dessa rotina constitui, à semelhança do já descrito para a receção, uma secção crítica. Nessa zona de código, deve ser efetuada a desativação das interrupções de transmissão da UARTx.

Transmissão – rotina de serviço à interrupção

Na UARTx do PIC32 uma interrupção é gerada enquanto o FIFO de transmissão tiver, pelo menos, uma posição livre[10]. Na estrutura do *device-driver,* a rotina de serviço à interrupção lê do *buffer* de transmissão o caracter referenciado pela variável **head** e escreve-o no registo de transmissão da UARTx. Adicionalmente, atualiza as variáveis **head** e **`count`** do *buffer* de transmissão e, quando a variável **`count`** atingir o valor 0 deve também desativar as interrupções de transmissão da UARTx. A ativação é efetuada na função **`comDrv_putc()`** sempre que se colocar um novo caracter no *buffer* circular.

[10] Este é um dos 4 modos de geração de interrupção possíveis no PIC32. Para mais detalhes deve ser consultado o manual da UART.

Trabalho a realizar

1. Escreva, as macros de ativação e desativação das interrupções de receção e de transmissão da UART1:

```
#define  DisableUart1RxInterrupt()    IEC0bits.U1RXIE = 0
#define  EnableUart1RxInterrupt()     ...
#define  DisableUart1TxInterrupt()    ...
#define  EnableUart1TxInterrupt()     ...
```

2. Declare a estrutura que implementa um *buffer* circular e crie duas instâncias dessa estrutura: **`rxb`** e **`txb`** (veja a introdução para mais detalhes). Defina a constante **`BUF_SIZE`** com o valor 8.
3. Escreva as funções **`comDrv_flushRx()`** e **`comDrv_flushTx()`** que inicializam os *buffers* circulares de transmissão e de receção:

```
void comDrv_flushRx(void)
{
   // Initialize variables of the reception buffer
}

void comDrv_flushTx(void)
{
   // Initialize variables of the transmission buffer
}
```

4. Escreva a função **`comDrv_putc()`** que escreve um caracter no *buffer* de transmissão e atualiza as variáveis **`tail`** e **`count`**. Esta função deverá ainda esperar que haja espaço livre no *buffer* antes de copiar um novo caracter:

```
void comDrv_putc(char ch)
{
   while(txb.count == BUF_SIZE); // Wait while buffer is full
   txb.data[txb.tail] = ch;      // Copy character to the transmission
                                 //  buffer at position "tail"
   txb.tail = (txb.tail + 1) & INDEX_MASK;  // Increment "tail" index
                                            //  (mod. BUF_SIZE)
   DisableUart1TxInterrupt();    // Begin of critical section
   // Increment "count" variable
   EnableUart1TxInterrupt();     // End of critical section
}
```

5. Escreva a função **`comDrv_puts()`** que evoca a função escrita no exercício anterior para enviar para a linha série uma *string* (terminada com o caracter '\0'):

```
void comDrv_puts(char *s)
{
   (...)
}
```

6. Escreva a rotina de serviço à interrupção de transmissão da UART1.

```
// If U1TXIF is set
   {
   // If "count" variable (transmission buffer, txb) is greater than 0
      {
         // Copy character pointed by "head" to U1TXREG register
         // Increment "head" variable (mod BUF_SIZE)
         // Decrement "count" variable
      }
      // If "count" variable is 0 then
            DisableUart1TxInterrupt();
      // Reset UART1 TX interrupt flag
   }
```

7. Escreva a função **`main()`** para testar as funções que escreveu nos pontos anteriores. Para a função **`comDrv_config()`** pode reaproveitar o código escrito anteriormente para a função **`configUart()`**.

```
int main(void)
{
   comDrv_config(115200,'N',1); // default "pterm" parameters
                                //  with TX and RX interrupts disabled
   comDrv_flushRx();
   comDrv_flushTx();
   EnableInterrupts();
   while(1)
      comDrv_puts("Teste do bloco de transmissao do device driver ");
}
```

8. Escreva a função **`comDrv_getc()`** que lê um caracter do *buffer* de receção. A função devolve o booleano **FALSE** se o número de caracteres no *buffer* for zero e **TRUE** no caso contrário. O caracter lido do *buffer* de receção deve ser passado através do ponteiro **`pchar`**:

```
char comDrv_getc(char *pchar)
{
   // Test "count" variable (reception buffer) and return FALSE
   //  if it is zero
   DisableUart1RxInterrupt();    // Begin of critical section
   // Copy character pointed by "head" to *pchar
   // Decrement "count" variable
   // Increment "head" variable (mod BUF_SIZE)
   EnableUart1RxInterrupt();     //  End of critical section
   return TRUE;
}
```

9. Escreva a rotina de serviço à interrupção de receção da UART1.

```
// If U1RXIF is set
   {
      rxb.data[rxb.tail] = U1RXREG;   // Read character from UART and
                                      //  write it to the "tail" position
                                      //  of the reception buffer
      // Increment "tail" variable (mod BUF_SIZE)
      // If reception buffer is not full (e.g. count < BUF_SIZE) then
      //   increment "count" variable
      // Else
      //   increment "head" variable (discard oldest character)
      // reset UART1 RX interrupt flag
   }
```

10. Re-escreva a função **`main()`** que escreveu no exercício 7 de modo a fazer o eco dos caracteres recebidos.

```
void main(void)
{
   comDrv_config(115200,'N',1); // default "pterm" parameters
                                //  with RX interrupts enabled and TX
                                //  interrupts disabled
   // (...)
   comDrv_puts("PIC32 UART Device-driver\n");
   while(1)
   {
      // Read character from reception buffer
      // Send character to the transmission buffer
   }
}
```

11. Altera a função anterior de modo a transmitir uma string com, pelo menos, 30 caracteres (à sua escolha) sempre que seja recebido o caracter 'S'.

12. A rotina de serviço à interrupção implementada no exercício 6 apenas transfere para a UART um caracter, de cada vez que é executada. O procedimento de transmissão pode, contudo, ser melhorado, se se copiar mais do que 1 caracter diminuindo, desse modo, o *overhead* resultante do processo de interrupção. Nesse sentido, altere a rotina de serviço à interrupção de transmissão de modo a copiar para a UART1 todos os caracteres do *buffer* circular de transmissão ou copiar até o FIFO da UART1 ficar cheio – a situação limite é a que ocorrer em primeiro lugar. Faça os testes que lhe permitam averiguar que as alterações efetuadas funcionam como pretendido.

13. Também a rotina de serviço à interrupção da receção pode ser melhorada, copiando para o *buffer* circular de receção mais do que um caracter de cada vez que é executada. Faça as alterações a essa rotina de modo a copiar para o *buffer* de receção, até ao limite de espaço disponível, todos os caracteres disponíveis no FIFO de receção da UART.

14. Acrescente mais um campo à estrutura do *buffer* circular de modo a passar a ser possível determinar se existiu *overrun* na receção de caracteres.

Elementos de apoio

- Slides das aulas teóricas.
- PIC32 Family Reference Manual, Section 08 – Interrupts.
- PIC32 Family Reference Manual, Section 21 – UART.
- PIC32MX5XX/6XX/7XX, Family Data Sheet, Pág. 122 a 124.

Trabalho prático N.º 10

Objetivos

- Compreender os mecanismos básicos que envolvem a comunicação série usando o protocolo I²C.
- Implementar funções básicas de comunicação série através do módulo I²C do PIC32 e utilizar essas funções para interagir com um sensor de temperatura com interface I²C.

Introdução

A interface I²C (acrónimo de *Inter-Integrated Circuit*) é uma interface de comunicação série bidirecional *half-duplex* pensada para interligação, a pequenas distâncias, entre microcontroladores e dispositivos periféricos. O PIC32, na versão usada na placa DETPIC32, disponibiliza 4 módulos I²C[11] que podem operar, cada um deles, como um dispositivo *slave*, como um dispositivo *master* num sistema com um único *master*, ou ainda como um dispositivo *master/slave* num sistema *multi-master*. Neste trabalho prático apenas iremos explorar a sua função como *master* num sistema com um único *master*. A Figura 20 apresenta o diagrama de blocos simplificado do módulo I²C do PIC32 onde se pretende, sobretudo, identificar os registos do modelo de programação e a sua função na perspectiva de utilização do módulo como único *master*.

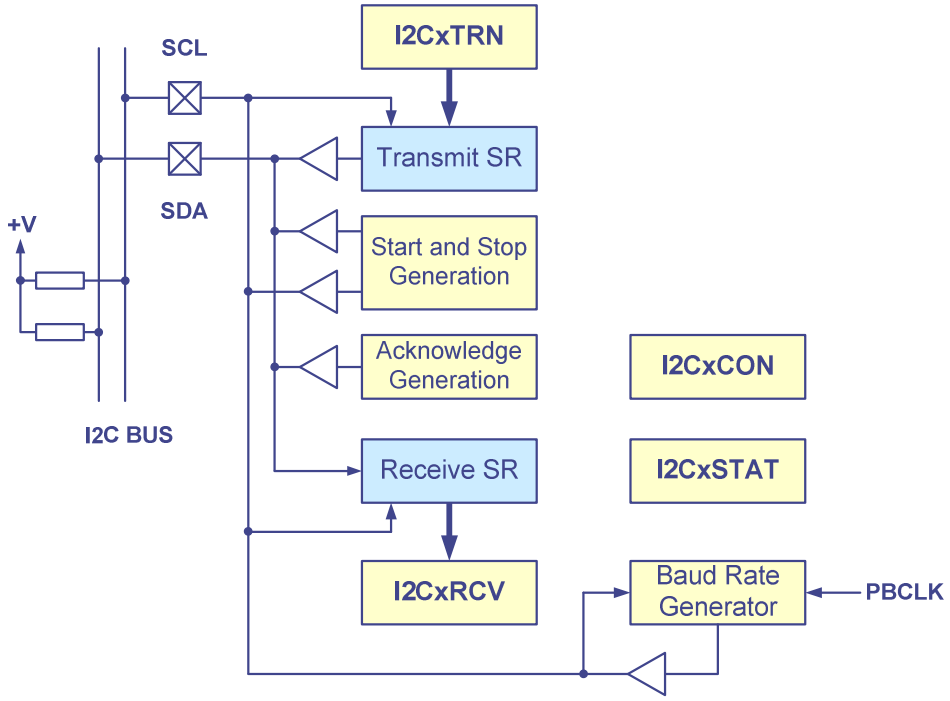


Figura 20. Diagrama de blocos simplificado do módulo I2C (*master*).

Os blocos-base do módulo I²C são dois *shift registers,* o *Transmit* SR e o *Receive* SR que fazem a conversão paralelo-série, e série-paralelo, respetivamente. Numa operação de transmissão, o *Transmit* SR envia para a linha SDA o *byte* armazenado no registo **`I2CxTRN`**. Por seu lado, numa operação de receção, o *byte* recebido no *Receive* SR é copiado para o registo **`I2CxRCV`**.

O módulo I²C limita-se a implementar as funções básicas que permitem a comunicação de acordo com o *standard*. A sequência de ações a realizar para a comunicação com um *slave* I²C é, assim, integralmente efetuada por *software*. Por exemplo, uma sequência de comunicação

[11] Esses 4 módulos são numerados pelo fabricante de 1 a 5: 1, 3, 4 e 5.

com um *slave* envolve, no início, o envio de um *start*, seguido de 8 bits com o endereço do *slave* e a operação a realizar (RD/WR\). Esta é também a sequência de operações a realizar no programa: envio do *start* através do registo **I2CxCON**, envio do *byte* com o endereço e a operação a realizar através do registo **I2CxTRN**.

O *Baud Rate Generator* é usado, na situação em que o módulo I^2C é *master*, para estabelecer a frequência do relógio na linha SCL. De acordo com a norma, esta frequência pode ser 100 kHz, 400 kHz ou 1 MHz, devendo ser estabelecida em função de cada um dos *slaves* com que o *master* vai comunicar.

Após *power-on* ou *reset* todos os 4 módulos I^2C estão inactivos. A ativação é efetuada através do bit ON do registo **I2CxCON**. A ativação de um dado módulo I^2C configura automaticamente os pinos correspondentes do PIC32 como *open-drain*, sobrepondo-se esta configuração à efetuada através do(s) registo(s) TRISx.

Gerador de *baudrate*

Como já referido anteriormente, o gerador de *baudrate* funciona como um gerador do relógio que é enviado para a linha SCL. A implementação deste gerador baseia-se num contador decrescente de 12 bits, em que o sinal de relógio é o PBCLK (*Peripheral Bus Clock*) cuja frequência é, na placa DETPIC32, 20 MHz. O contador tem uma entrada de *load* síncrona, que permite carregar o valor inicial de contagem. Adicionalmente, o contador tem uma saída de *terminal count* que fica ativa sempre que o contador atinge o valor 0. Uma vez atingido o valor 0, o contador só retoma o funcionamento quando for efetuado o *reload* do valor inicial. O valor incial de contagem é armazenado no registo **I2CxBRG**, registo este que assim determina a frequência-base do relógio na linha SCL. A Figura 21 apresenta um diagrama simplificado do gerador de *baudrate*, que não toma em consideração as questões relacionadas com a sincronização do relógio (*clock stretching*).

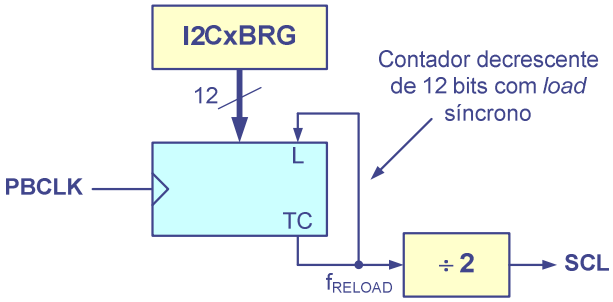


Figura 21. Diagrama de blocos simplificado do gerador de *baudrate*.

O facto de o contador efetuar o *reload* do valor inicial de forma síncrona implica que a combinação 0 (que provoca a ativação da saída TC) esteja fixa durante 1 ciclo de relógio do sinal PBCLK (sem *clock stretching*). Sendo assim, e de forma análoga ao já discutido para os *timers*, a frequência do sinal na saída TC (f_{RELOAD}) é dada por:

$$f_{RELOAD} = f_{PBCLK} / (I2CxBRG + 1)$$

A frequência do sinal à saída do contador é posteriormente dividida por 2 (utilizando, por exemplo, um *flip-flop* tipo T), de modo a obter-se um sinal com um *duty-cycle* de 50%. Assim, a frequência do sinal de relógio na linha SCL é:

$$f_{SCL} = f_{PBCLK} / (2 * (I2CxBRG + 1))$$

O valor da constante a colocar no registo **I2CxBRG** fica então:

I2CxBRG $= f_{PBCLK} / (2 * f_{SCL}) - 1$, ou melhor ainda (com arredondamento):

I2CxBRG $= (f_{PBCLK} + f_{SCL}) / (2 * f_{SCL}) - 1$

Configuração do módulo I^2C

Para a configuração do módulo I^2C bastam duas ações:

- Configuração do gerador de *baudrate*: cálculo da constante I2CxBRG e escrita no respetivo registo.
- Activação do módulo I^2C (bit ON do registo **`I2CxCON`**).

Programação com o módulo I^2C

O módulo I^2C suporta as funções básicas de comunicação através de geradores de *start* e de *stop*, transmissão de um *byte*, receção de um *byte* e geração de *acknowledge*. Em termos gerais, a programação de cada um dos passos do protocolo consiste em duas operações básicas: 1) escrita de um registo (de dados ou de controlo em função do passo específico); 2) espera até que a operação se realize (*polling* de um registo de um bit ou conjunto de bits de um registo). Este procedimento tem que ser seguido em todas as operações básicas, uma vez que o módulo I^2C não admite o começo de uma nova operação sem que a anterior tenha terminado. Por exemplo, não é possível enviar um *start* e, logo de seguida escrever no registo **`I2CxTRN`** para iniciar a transmissão de um *byte*, sem esperar que a operação de *start* termine. Se esta regra não for seguida, o módulo ignora, neste caso, a escrita no registo **`I2CxTRN`**, e ativa o bit **`IWCOL`** (*write collision detect bit*) do registo **`I2CxSTAT`**.

Geração do *start*

Para iniciar um evento de *start* activa-se o bit **`SEN`** (*start enable bit*) do registo **`I2CxCON`**. Por *hardware*, este bit passa automaticamente a zero quando a operação é completada. Este bit pode, assim, ser usado, por *polling*, para determinar a passagem para a operação seguinte.

Transmissão de um *byte* para um *slave*

A transmissão de um *byte* de dados ou de um endereço para um *slave* é efetuada escrevendo o valor a enviar no registo **`I2CxTRN`**. O envio desse valor demora 8 ciclos de relógio (SCL) e no 9º ciclo o *master* lê da linha o valor do bit de *acknowledge* colocado pelo *slave*. Esse valor é colocado no bit **`ACKSTAT`** do registo **`I2CxSTAT`**. Para garantir que esta sequência chegou ao fim, e assim passar para outra operação, pode fazer-se o *polling* do bit **`TRSTAT`** (*transmission status bit*) do registo **`I2CxSTAT`**: enquanto a sequência decorre esse bit está a 1, passando a 0 após o 9º ciclo de relógio, isto é, após a receção do bit de *acknowledge*.

O bit **`ACKSTAT`** reflecte directamente o valor recebido da linha (que representa ACK\): esse bit está a 0 se o *slave* recebeu corretamente o valor e fica a 1, no caso contrário.

Receção de um *byte* transmitido por um *slave*

O *master* pode receber informação do *slave* após ter enviado uma sequência com o endereço e o bit R/W\ a 1. Para isso, o *master* tem que ativar o bit **`RCEN`** (*receive enable bit*) do registo **`I2CxCON`**. No entanto, antes de ativar esse bit, é necessário garantir, de acordo com as indicações do fabricante, que os 5 bits menos significativos do registo **`I2CxCON`** são todos 0 (ver manual do I^2C do PIC32).

Após a ativação do bit **`RCEN`**, o *master* inicia a geração do relógio na linha SCL e, após 8 ciclos desse relógio, o *shift-register* de receção do *master* recebeu o valor enviado pelo *slave.* Após o 8º ciclo de relógio o bit **`RCEN`** é automaticamente desativado, o *byte* recebido no *shift-register* é copiado para o registo **`I2CxRCV`** e o bit **`RBF`** (*receive buffer full status bit*) do registo **`I2CxSTAT`** é ativado. Este bit pode ser usado por *polling* para esperar que o *byte* seja recebido.

Transmissão do *acknowledge*

Após a receção de um *byte*, o *master* tem que transmitir uma sequência de *acknowledge* (ACK\): transmite um 0 (ACK\=0), no caso em que pretende continuar a ler informação do *slave*; transmite um 1 (ACK\=1, i.e., NACK), no caso em que pretende terminar a comunicação.

O bit **`ACKDT`** (*acknowledge data bit*) do registo **`I2CxCON`** permite especificar ACK\ (0) ou NACK (1). O bit **`ACKEN`** (*acknowledge sequence enable bit*), quando ativado, inicia a sequência de *acknowledge*. Este bit é automaticamente desativado pelo *hardware* quando o *master* termina o envio da sequência de *acknowledge*, pelo que pode ser usado, por *polling*, para determinar a passagem para a operação seguinte.

Geração do *stop*

Antes de se iniciar um evento de *stop* é necessário garantir que os 5 bits menos significativos do registo **`I2CxCON`** são todos 0. Quando essa condição é verificada pode então gerar-se um evento de *stop* através da ativação do bit **`PEN`** (*stop enable bit*) do registo **`I2CxCON`**. À semelhança do referido para os restantes eventos, deve também esperar-se que a ação *stop* termine antes de prosseguir com outro qualquer evento. Para isso basta esperar que o bit **`PEN`** passe ao nível lógico 0, uma vez que ele é automaticamente colocado a 0 pelo *hardware* quando a ação termina.

Sensor de temperatura TC74

O circuito integrado TC74 da Microchip é um sensor digital de temperatura com interface série I^2C. O elemento sensorial integrado no dispositivo permite a medida de temperatura na gama -65ºC a 125ºC, com uma precisão de ±2ºC na gama 25ºC a 85ºC. O valor da temperatura é disponibilizado num registo interno de 8 bits do sensor, e é codificado em complemento para 2 (há, portanto, necessidade de se fazer a extensão de sinal se usar uma variável tipo **`int`**).

O endereço do dispositivo, para efeitos da sua interligação num barramento I^2C, é fixado pelo fabricante, havendo no mercado versões do mesmo sensor com 7 endereços distintos. O sensor disponível para ser usado nestas aulas tem o endereço 0x4D (endereço de 7 bits – ver página 13 do manual do sensor de temperatura).

O sensor de temperatura tem internamente dois registos: o já mencionado registo de temperatura (designado por registo **`TEMP`**), que apenas pode ser lido e o registo de configuração (designado por registo **`CONFIG`**) que pode ser lido ou escrito. Para o acesso a estes dois registos são disponibilizados dois comandos: o comando **`RTR`** (*read temperature*) que permite a leitura do registo **`TEMP`** e o comando **`RWCR`** (*read/write configuration*) que permite a escrita ou a leitura do registo **`CONFIG`**. A Figura 22 apresenta o protocolo I^2C para a leitura de um registo interno do sensor de temperatura (ver página 7 do manual do sensor).

Read Byte Format

S	Address	WR	ACK	Command	ACK	S	Address	RD	ACK	Data	NACK	P
	7 Bits			8 Bits			7 Bits			8 Bits		

Slave Address

Command Byte: selects which register you are reading from.

Slave Address: repeated due to change in data-flow direction.

Data Byte: reads from the register set by the command byte.

Figura 22. Protocolo I^2C para leitura de 1 *byte* de um registo interno do sensor de temperatura.

A sequência de ações que o programa tem que levar a cabo para a leitura de um registo do sensor é a que resulta da descrição do protocolo apresentada na figura anterior. Se se pretender ler o valor da temperatura, a sequência de ações a efetuar é:

1) Enviar um *start.*
2) Enviar um *byte* com o endereço do sensor e com a indicação de um operação de escrita (R/W\ = 0).
3) Esperar pela receção do *acknowledge* do sensor.
4) Enviar um *byte* com a identificação do comando RTR (valor 0x00 – ver manual).
5) Esperar pela receção do *acknowledge* do sensor.
6) Enviar um *start* – este novo *start* tem que ser enviado porque a operação que se vai especificar de seguida é diferente da anterior (a anterior foi uma escrita, a seguinte vai ser uma leitura).
7) Enviar um *byte* com o endereço do sensor e com a indicação de um operação de leitura (R/W\ = 1).
8) Esperar pela receção do *acknowledge* do sensor.
9) Ativar o bloco de receção do *master*.
10) Esperar que o *master* receba o *byte* do *slave*.
11) Enviar um *not acknowledge* (NACK) sinalizando-se desse modo o *slave* que o *master* não pretende continuar a ler.
12) Enviar um *stop*.

Trabalho a realizar

Parte I

1. Monte, na placa branca, o sensor de temperatura TC74, de acordo com a figura seguinte. Os sinais SDA1 (RD9) e SCL1 (RD10) correspondem à ligação ao módulo I^{2}C número 1 do PIC32[12].

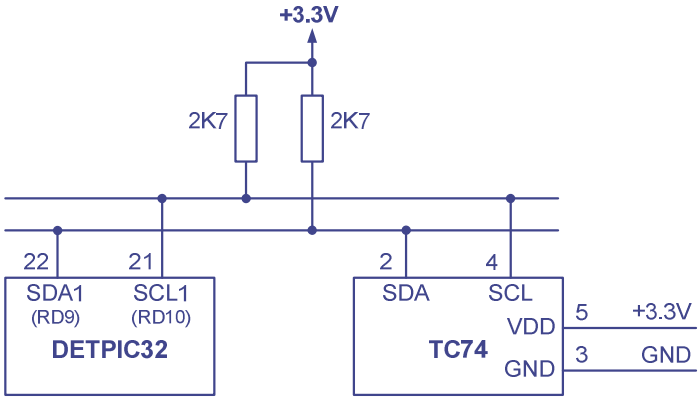


Figura 23. Ligação do sensor de temperatura TC74 à placa DETPIC32.

2. A abordagem que vamos seguir neste trabalho prático é a de implementar funções específicas para cada um dos eventos do protocolo I^{2}C. Tendo essas funções-base implementadas, a comunicação com um *slave* resume-se à evocação dessas funções pela ordem adequada, de acordo com a operação que se pretende realizar. Assim, comece por escrever uma função para inicialização do módulo I^{2}C:

```
void i2c1_init(unsigned int clock_freq)
{
   // Config baudrate generator (see introduction for details)
   // Enable I2C1 module
}
```

[12] O PIC32 disponibiliza 4 módulos I^{2}C iguais. No entanto, devido a um problema de *hardware* reportado pela Microchip no documento "PIC32MX7XX Family Errata", os módulos 4 e 5 requerem um procedimento de configuração inicial diferente dos restantes dois.

3. Implemente a função para gerar o evento de *start*. Essa função deve implementar a seguinte sequência de ações: 1) ativar o *start*; 3) esperar que o evento termine.

```
void i2c1_start(void)
{
   // Activate Start event (I2C1CON, bit SEN)
   // Wait for completion of the Start event (I2C1CON, bit SEN)
}
```

4. Implemente a função para gerar o evento de *stop*. Essa função deve implementar a seguinte sequência de ações: 1) assegurar-se que os 5 bits menos significativos do registo **I2CxCON** são todos 0; 2) ativar o *stop*; 3) esperar que o evento termine.

```
void i2c1_stop(void)
{
   // Wait until the lower 5 bits of I2CxCON are all 0 (the lower 5 bits
   //   of I2CxCON must be 0 before attempting to set the PEN bit)
   // Activate Stop event (I2C1CON, bit PEN)
   // Wait for completion of the Stop event (I2C1CON, bit PEN)
}
```

5. Implemente a função para transmitir um *byte*. Essa função deve implementar a seguinte sequência de ações: 1) copiar para o registo **I2C1TRN** o *byte* a transmitir; 2) esperar que o *byte* seja transmitido e que o *acknowledge* seja recebido. A função deverá retornar o valor de *acknowledge* recebido.

```
int i2c1_send(unsigned char value)
{
   // Copy "value" to I2C1TRN register
   // Wait while master transmission is in progress (8 bits + ACK\)
   //    (I2C1STAT, bit TRSTAT – transmit status bit)
   // Return acknowledge status bit (I2C1STAT, bit ACKSTAT)
}
```

6. Implemente a função para receber um *byte*. Essa função deve implementar a seguinte sequência de ações: 1) esperar que os 5 bits menos significativos do registo I2C1CON sejam todos 0; 2) ativar o bit **RCEN** (registo **I2C1CON**); 3) esperar que o *byte* seja recebido (**I2C1STAT** bit **RBF**); 4) enviar ACK ou NACK e esperar que a sequência termine; 5) retornar o valor do registo **I2C1RCV**.

```
char i2c1_receive(char ack_bit)
{
   // Wait util the lower 5 bits of I2C1CON are all 0 (the lower 5 bits
   //   of I2C1CON must be 0 before attempting to set the RCEN bit)
   // Activate RCEN bit (receive enable bit, I2C1CON)
   // Wait while byte not received (I2C1STAT, bit RBF – receive buffer
   //   full status bit)
   // Send ACK / NACK bit. For that:
   //   1. Copy "ack_bit" to I2C1CON, bit ACKDT (be sure "ack_bit" value
   //      is only 0 or 1)
   //   2. Start Acknowledge sequence (I2C1CON, bit ACKEN)
   // Wait for completion of Acknowledge sequence (I2C1CON, bit ACKEN)
   // Return received value (I2C1RCV)
}
```

7. Até ao exercício anterior implementámos as funções necessárias para lidar com todos os eventos I^2C necessários para estabelecer a comunicação com um *slave*. A comunicação com o sensor para a leitura do valor da temperatura resume-se à evocação das funções anteriores pela ordem definida no protocolo. Assim, implemente a função **main()** para ler 4

valores de temperatura por segundo – os valores lidos devem ser impressos no ecrã usando *system calls* (verifique na Figura 22 a sequência de ações a realizar). Poderá utilizar sequências de *escape* (ver parte final do ficheiro "detpic32.h") para melhorar a aparência (por exemplo, impressos sempre no mesmo sítio) dos valores impressos no ecrã.

```
#define  I2C_READ       1
#define  I2C_WRITE      0
#define  I2C_ACK        0
#define  I2C_NACK       1

#define  ADDR_WR        ((SENS_ADDRESS << 1) | I2C_WRITE)
#define  ADDR_RD        ((SENS_ADDRESS << 1) | I2C_READ)
#define  SENS_ADDRESS   0x4D    // device dependent
#define  TC74_CLK_FREQ  100000  // 100 KHz
#define  RTR            0       // Read temperature command

int main(void)
{
   int ack, temperature;
   i2c1_init(TC74_CLK_FREQ);
   while(1)
   {
      // Send Start event
      // Send Address + WR (ADDR_WR); copy return value to "ack" variable
      // Send Command (RTR); add return value to "ack" variable
      // Send Start event (again)
      // Send Address + RD (ADDR_RD); add return value to "ack" variable
      // Test "ack" variable; if "ack" != 0 then an error has occurred;
      //     send the Stop event, print an error message and exit loop
      // Receive a value from slave (send NACK as argument); copy
      //     received value to "temperature" variable
      // Send Stop event
      // Print "temperature" variable (syscall printInt10)
      // Wait 250 ms
   }
}
```

Parte II

1. Neste exercício pretende-se a implementação de um módulo genérico com as funções-base de interação com o módulo I^2C, de modo a poder ser facilmente incluído num projeto de maior dimensão. Assim:

   - Construa o ficheiro "**i2c.h**", com a estrutura-base que a seguir se apresenta, e onde estejam declarados os protótipos de todas as funções de interface com o módulo I^2C que desenvolveu na parte 1 deste trabalho prático, bem como todos os símbolos gerais que tenham que ser usados (**I2C_READ**, **I2C_WRITE**, ...).

```
// i2c.h
#ifndef I2C_H
#define I2C_H

// Declare symbols here (READ, WRITE, ...)
(...)

// Declare function prototypes here
(...)

#endif
```

- Construa o ficheiro "**i2c.c**" com o código de cada uma das funções de interação com o módulo I^2C.

```
// i2c.c
#include <detpic32.h>
#include "i2c.h"

(...)
```

2. Escreva uma função que efetue a leitura de um valor de temperatura do sensor, usando as funções de comunicação que desenvolveu anteriormente e que estão disponíveis no ficheiro "**i2c.c**". Escreva a função **main()** para teste desta função, de modo a fazer 4 leituras por segundo e a imprimir no ecrã o respectivo valor de temperatura (utilize *system calls*).

```
#include <detpic32.h>
#include "i2c.h"

(...)

int getTemperature(int *temperature)
{
   int ack;
   // Send Start event
   // Send Address + WR (ADDR_WR) and copy return value to "ack" variable
   ...
   (see exercise 7, function main())
   ...
   // Send Stop event
   return ack;
}
```

Nota:

Para compilar o projeto com mais do que um ficheiro pode usar o *script* "**pcompile**" colocando a lista dos ficheiros que fazem parte do projeto separados por um espaço. Exemplo:

```
pcompile guiao10.c i2c.c
```

O comando anterior compila os dois ficheiros e, se não houver erros de sintaxe, efetua o *link* e gera um ficheiro com o nome do primeiro ficheiro substituindo a extensão "**.c**" pela extensão "**.hex**". Para o exemplo acima, o ficheiro de saída seria "**guiao10.hex**".

3. Retome agora o código que desenvolveu no último exercício do trabalho prático n.º 7 e acrescente a função de leitura da temperatura que escreveu no exercício anterior. Faça as alterações ao programa que permitam mostrar nos dois *displays* o valor da temperatura, sempre que a combinação binária nos bits RE5 e RE4 seja "11". A leitura do sensor deverá ser feita por interrupção, 4 vezes por segundo.

Elementos de apoio

- Slides das aulas teóricas.
- Tiny Serial Digital Thermal Sensor TC74 (disponível no site da disciplina).
- PIC32 Family Reference Manual, Section 24 – I2C.

Trabalho prático N.º 11

Objetivos

- Compreender os mecanismos básicos que envolvem a programação série usando o protocolo SPI.
- Implementar funções de comunicação no PIC32 que permitam a interação com uma memória EEPROM com interface SPI.

Introdução

A interface SPI (acrónimo de *Serial Peripheral Interface*) é uma interface de comunicação série bidirecional *full-duplex* vocacionada para ligações de pequena distância entre periféricos e microcontroladores. É uma interface de comunicação síncrona com relógio explícito do *master*, isto é, o relógio é gerado pelo *master* que o disponibiliza para todos os *slaves*. Utiliza uma arquitetura *master-slave* com ligação ponto a ponto que funciona em modo "data exchange": por cada bit que é enviado para o *slave* é recebido 1 bit. Assim, ao fim de N ciclos de relógio o *master* enviou uma palavra de N bits e recebeu, do *slave*, uma palavra também com N bits. A título de exemplo, se o *master* pretender ler do *slave* uma palavra de 16 bits, tem que transmitir uma palavra de 16 bits (que será descartada pelo *slave* se de facto se tratar apenas de um operação de leitura). A Figura 24 mostra, de forma esquemática, o funcionamento da interface SPI.

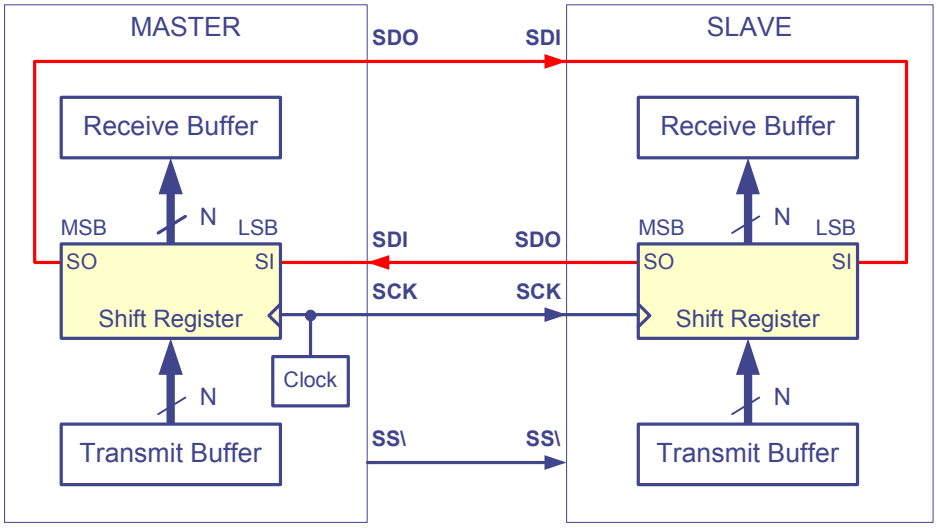


Figura 24. Esquema de princípio da comunicação série SPI.

O PIC32, na versão usada na placa DETPIC32, disponibiliza 3 módulos SPI, numerados pelo fabricante como SPI2, SPI3 e SPI4. A Figura 25 apresenta o diagrama de blocos simplificado do módulo SPI. Os elementos-base de cada um desses módulos são: *shift-register* (embora não representado na figura, existem dois *shift-registers* separados – um para receção e outro para transmissão), um FIFO de transmissão, um FIFO de recepção e um gerador de *baudrate*.

O comprimento de palavra a usar na comunicação é configurável, podendo ser 8, 16 ou 32 bits. O número de posições de cada FIFO é dependente do comprimento de palavra selecionado, sendo de 16 posições se o comprimento de palavra for 8 bits, de 8 posições se o comprimento de palavra for 16 bits e de 4 posições se o comprimento de palavra for 32 bits.

O acesso aos FIFOs de transmissão e de receção é efetuado através do registo **`SPIxBUF`**: uma escrita neste registo traduz-se no acesso indireto ao FIFO de transmissão, enquanto que a leitura desse registo traduz-se num acesso ao FIFO de receção.

O *Baudrate generator* gera o sinal de relógio que é enviado, através da linha **`SCK`**, para todos os *slaves* do sistema. Este gerador apenas é ativado quando o módulo é configurado como

master. Além disso, o sinal de relógio só é gerado quando há comunicação, estando num estado de repouso (configurável) quando o módulo não está a transmitir informação.
Após *power-on* ou *reset* todos os 3 módulos SPI estão inativos. A ativação é efetuada através do bit ON do registo **`SPIxCON`**. A ativação de um dado módulo SPI configura automaticamente os pinos correspondentes do PIC32 como entrada ou saída, consoante os casos, sobrepondo-se esta configuração à efetuada através do(s) registo(s) **`TRISx`**.

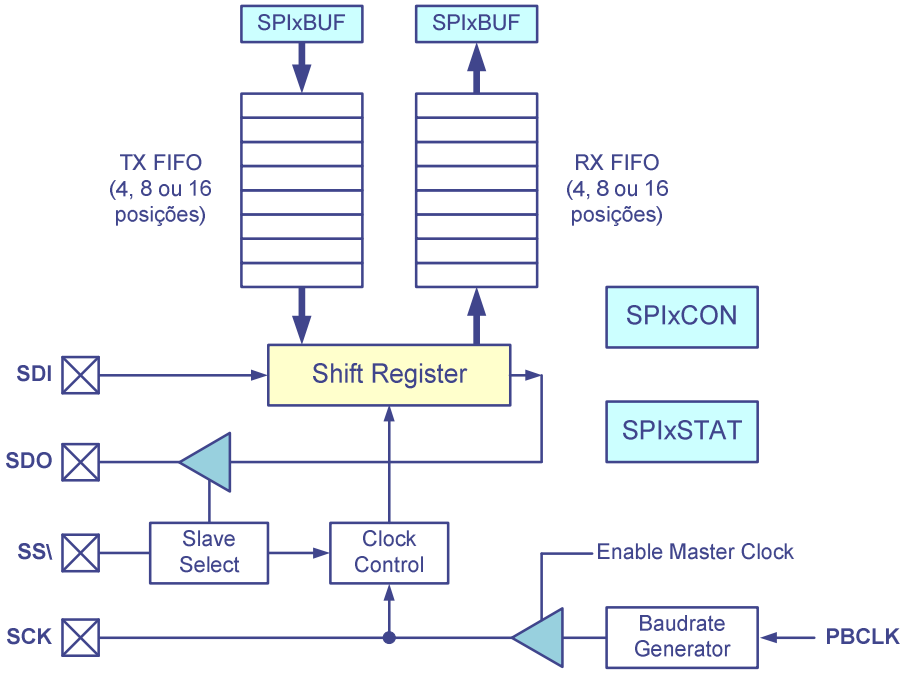


Figura 25. Diagrama de blocos simplificado do módulo SPI do PIC32.

Gerador de *baudrate*

O gerador de *baudrate* utiliza uma arquitetura semelhante à de um *timer*, em que o sinal de relógio de entrada é o *Peripheral Bus Clock* (20 MHz na placa DETPIC32). Dada a obrigatoriedade, imposta pelo funcionamento da interface SPI, de o relógio ter um *duty-cycle* de 50%, este bloco inclui, à saída do comparador, um divisor por 2.

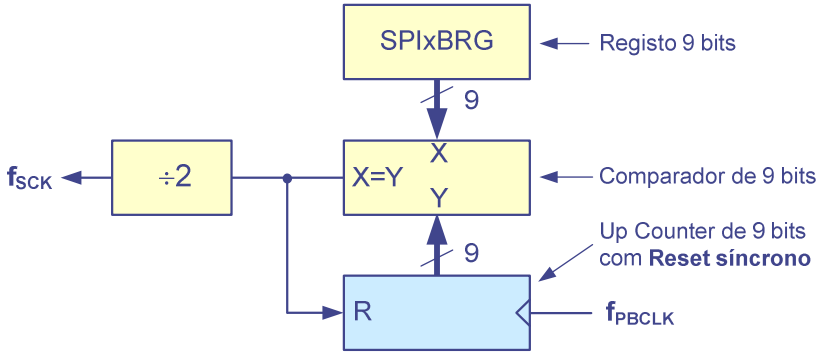


Figura 26. Diagrama de blocos do gerador de *baudrate.*

A frequência à saída deste módulo é, então,

$$f_{SCK} = f_{PBCLK} / (2 * (SPIxBRG + 1))$$

pelo que o valor (arredondado) de **`SPIxBRG`** é dado por:

$$SPIxBRG = (f_{PBCLK} + f_{SCK}) / (2 * f_{SCK}) - 1$$

em que **SPIxBRG** representa a constante armazenada no registo com o mesmo nome. De notar que o valor máximo da constante de divisão é 512 (o *timer* tem uma arquitetura de 9 bits).

Configuração do módulo SPI

A configuração do módulo SPI (como *master*) é, no essencial, efetuada de acordo com o *slave* com que se vai comunicar. Numa aplicação típica com vários *slaves* o módulo SPI é reconfigurado antes de se iniciar a comunicação com um dado *slave* tendo em conta as suas características específicas. Há, assim, um conjunto de ações de configuração gerais e um conjunto de acções de configuração que dependem das características do *slave*. As ações de configuração que dependem das características do *slave* são:

- Configurar o gerador de *baudrate*: cálculo da constante **SPIxBRG** e escrita no respetivo registo. A frequência máxima de relógio é um parâmetro intrínseco de cada *slave*.
- Configurar o nível lógico do relógio a que corresponde a situação de repouso.
- Configurar a transição ativa do relógio, isto é, em que há transmissão de dados: transição do estado ativo para o estado de repouso, ou o contrário.
- Configurar o instante em que o *master* armazena a informação recebida da linha.
- Configurar o comprimento de palavra: pode ser 8, 16 ou 32 bits.

As ações de configuração gerais, não dependentes do *slave*, são:

- Configurar o módulo como *master*.
- Ativar a utilização dos FIFOs de transmissão e receção (é possível a utilização do módulo SPI sem estes FIFOs)
- Ativar a utilização da linha *slave select* (**SS**) que permite selecionar automaticamente o *slave* durante a comunicação. Esta opção é útil quando o sistema apenas tem 1 *slave*. Se não for esse o caso, a linha de seleção tem que ser controlada por *software*, tipicamente através de um porto de saída por cada *slave*.
- Limpar o FIFO de receção.
- Limpar a *flag* de *overflow* na receção (se esta *flag* ficar activa o módulo descarta todas as palavras recebidas).
- Ativar o módulo SPI.

Todas estas configurações têm que ser efetuadas com o módulo desativado, pelo que essa deve ser a primeira ação do procedimento de configuração.

Programação com o módulo SPI

O módulo SPI realiza, no essencial, uma única operação: transmissão de uma palavra para o *slave* que estiver selecionado. Um vez que o sistema funciona em modo "data exchange", a transmissão de uma palavra de **n** bits envolve sempre a receção simultânea de uma palavra com a mesma dimensão (como referido anteriormente, a dimensão da palavra é configurável). Assim, o envio de um byte para o *slave* é feito através da cópia desse valor para o FIFO de transmissão através do registo **SPIxBUF**, podendo o valor recebido ser ignorado (a transmissão começa logo que o valor é copiado para o FIFO). Por outro lado, para a leitura de um byte de um *slave* é necessário transmitir um byte (dependendo das situações, o valor desse byte pode não ser relevante). Por cada bit enviado pelo *master* o *slave* transmite também um bit, pelo que após 8 ciclos de relógio o *master* recebe o byte. Esse byte recebido pelo *shift-register* é depois copiado para o FIFO de receção de onde pode ser lido pelo programa através do registo **SPIxBUF**.

Um aspeto importante a ter em consideração na programação com o módulo SPI é garantir que este termina todas as ações de transferência em curso, antes de passar para qualquer outra operação. O bit **SPIBUSY** (*SPI Activity Status bit*) do registo **SPIxSTAT** (ver página 23-9 do manual) fornece essa indicação.

Tendo estes aspetos em consideração, a programação com SPI é, essencialmente, orientada para as funções de interação e para o protocolo característicos de um dado dispositivo *slave*. Neste trabalho prático vai ser explorado o modo de funcionamento e programação da interface SPI do PIC32, usando como dispositivo *slave* uma memória EEPROM de 512 bytes (512x8). Na secção seguinte faz-se uma breve descrição dessa memória, cobrindo, essencialmente, a estrutura interna, o modo de operação e o protocolo de comunicação associado a cada uma das operações que é possível realizar. Esta descrição **não dispensa** a consulta do manual do fabricante que se encontra disponível no site da disciplina.

Memória EEPROM 25LC040A

Descrição geral

O circuito integrado 25LC040A da Microchip é uma memória de tecnologia não volátil EEPROM (*Electrically Erasable Programmable Read Only Memory*) com uma capacidade de 512 bytes (endereços **0x000** a **0x1FF**) e interface de comunicação série SPI. O comprimento de palavra, para qualquer operação sobre a memória, é de 8 bits.

O *shift-register* interno da interface SPI da memória recebe um novo bit da linha SI na transição ascendente do relógio e envia um novo bit para a linha SO na transição descendente seguinte (ver Figura 28). Este comportamento determina desde logo o modo como o relógio do *master* deve ser configurado, do ponto de vista da escolha das transições ativas, devendo ser compatível com o modo de funcionamento da memória. Assim, se a memória usa, como transição ativa do relógio para receção, a transição ascendente, o *master* deve usar como transição ativa para transmissão a transição descendente. O mesmo raciocínio deve ser seguido para a configuração da transição ativa de receção no *master*.

A componente de armazenamento da memória está organizada como uma matriz com 32 linhas e 16 colunas, sendo cada ponto da matriz constituído por 8 células de armazenamento de 1 bit (i.e. cada ponto da matriz corresponde a um ponto de armazenamento de 1 byte de informação). Assim, a componente de armazenamento pode ser vista como uma matriz de 32x16 bytes, sendo que cada linha dessa matriz armazena um total de 16 bytes. O número de palavras (bytes, neste caso) armazenado numa linha da matriz designa-se por página.

A funcionalidade completa da memória pode ser explorada através de 6 comandos distintos, sumariamente apresentados na Figura 27 (ver página 7 do manual da memória).

TABLE 2-1: INSTRUCTION SET

Instruction Name	Instruction Format	Description
READ	0000 $A_8$011	Read data from memory array beginning at selected address
WRITE	0000 $A_8$010	Write data to memory array beginning at selected address
WRDI	0000 x100	Reset the write enable latch (disable write operations)
WREN	0000 x110	Set the write enable latch (enable write operations)
RDSR	0000 x101	Read STATUS register
WRSR	0000 x001	Write STATUS register

Note: A_8 is the 9th address bit, which is used to address the entire 512 byte array.

Figura 27. Comandos de interação com a memória EEPROM.

Tal como representado na figura anterior, os 3 bits menos significativos do *instruction format* definem a operação a realizar. No caso de a operação ser uma leitura ou uma escrita o bit 3 (4º

bit menos significativo) é usado para enviar o bit mais significativo do endereço da memória (bit A_8).

Sequência de leitura de uma posição de memória

O protocolo com a descrição da sequência de operações a realizar para desencadear a leitura de uma posição de memória está representado, sob a forma de um diagrama temporal, na Figura 28 (ver também página 7 do manual). Assim, da análise dessa figura pode verificar-se que, em primeiro lugar, é enviado o byte de comando (**READ**, ver Figura 27) que inclui o bit mais significativo do endereço (A_8), seguido do byte representativo dos 8 bits menos significativos do endereço. Para receber o valor armazenado na correspondente posição de memória o *master* tem ainda que enviar um byte adicional, cujo valor é irrelevante e que será ignorado pela memória.

FIGURE 2-1: READ SEQUENCE

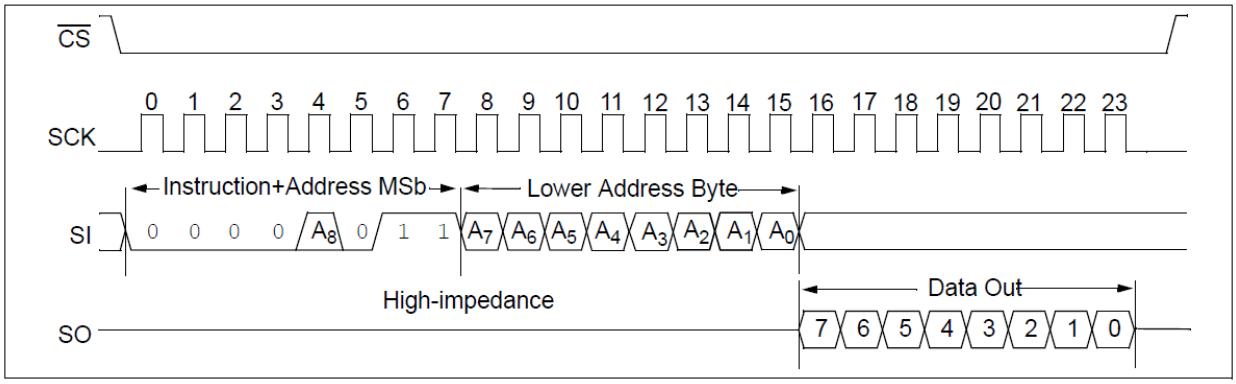


Figura 28. Sequência de leitura de uma posição de memória.

Uma operação de leitura é ignorada pela memória se ainda estiverem a decorrer operações internas relacionadas com uma escrita. De notar que o fim efetivo de uma operação de escrita não coincide com o fim das ações de comunicação. Ou seja, após ter terminado o processo de comunicação, a memória necessita de tempo adicional para consumar a operação. Esse tempo é destinado a: 1) fazer o apagamento da posição onde se pretende efetuar a escrita e 2) efetuar a escrita do byte recebido.

A memória disponibiliza um registo de 8 bits com as funções de controlo e de *status*, designado pelo fabricante por registo **STATUS**, que permite, entre outras coisas obter informação sobre a existência de um processo de escrita em curso. A Figura 29 representa a estrutura desse registo, onde se pode verificar que apenas os 4 bits menos significativos têm informação útil (ver também página 10 do manual).

TABLE 2-2: STATUS REGISTER

7	6	5	4	3	2	1	0
–	–	–	–	W/R	W/R	R	R
X	X	X	X	BP1	BP0	WEL	WIP
W/R = writable/readable. R = read-only.							

Figura 29. Estrutura do registo STATUS da memória.

Em particular, o bit menos significativo (**WIP**, *write in progress*) indica, quando a 1, que a memória está ocupada num operação de escrita, ou seja, que a última operação de escrita ainda está em curso. Assim, uma operação de leitura ou escrita deve sempre ser precedida da verificação (por *polling*) do estado deste bit através da leitura do registo **STATUS** (ver Figura 27). Faz-se, mais à frente, a descrição da sequência para efetuar a leitura desse registo.

Sequência de escrita de uma posição de memória

A Figura 30 apresenta, sob a forma de um diagrama temporal a sequência de operações para efetuar a escrita de uma posição de memória (ver também página 8 do manual). Tal como para a leitura, em primeiro lugar é enviado o byte de comando (**WRITE**, ver Figura 27) que inclui o bit mais significativo do endereço, seguido do byte menos significativo do endereço e, finalmente, do byte a escrever.

FIGURE 2-2: BYTE WRITE SEQUENCE

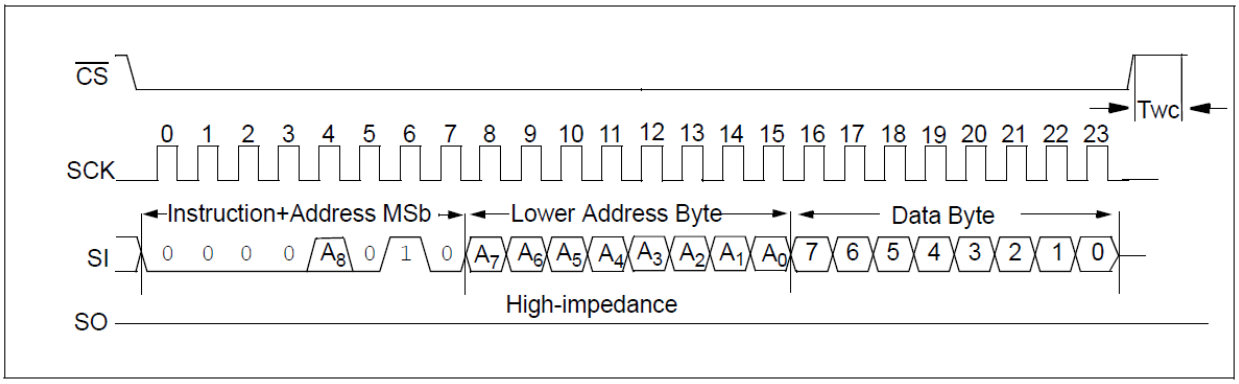


Figura 30. Sequência de escrita de uma posição de memória.

De modo a evitar a alteração acidental da informação armazenada, a memória EEPROM está sempre protegida contra operações de escrita. Ou seja, antes de efetuar uma sequência de escrita é sempre necessário desproteger a memória. A desproteção é efetuada através do envio do comando de ativação da escrita (**WREN**, ver Figura 27) que ativa o bit **WEL** (*write enable latch*) do registo **STATUS**.

Após o fim de uma operação de escrita (uma operação de escrita termina, do ponto de vista da comunicação, quando o sinal **CS** é desativado) a memória regressa ao estado de escrita protegida, pelo que, para cada operação de escrita é sempre necessário efetuar, em primeiro lugar, a desproteção.

Sequência de leitura do registo de STATUS

A Figura 31 descreve a sequência para ler o registo de **STATUS** da memória (ver também página 10 do manual). A sequência começa com o envio do comando respetivo (**RDSR**, ver Figura 27), seguido do envio de um byte, cujo valor é irrelevante, para permitir à EEPROM enviar o conteúdo do registo.

FIGURE 2-6: READ STATUS REGISTER TIMING SEQUENCE (RDSR)

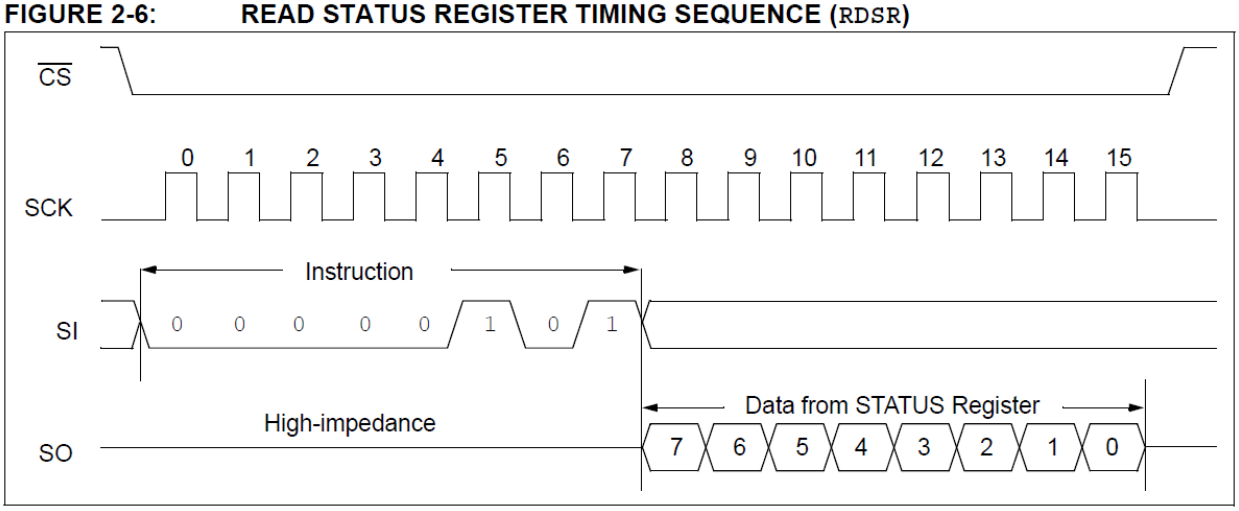


Figura 31. Sequência de leitura do registo de STATUS

O registo **STATUS** pode ser lido em qualquer momento independentemente de estarem ou não a ser realizadas outras operações internas.

Escrita no registo de STATUS

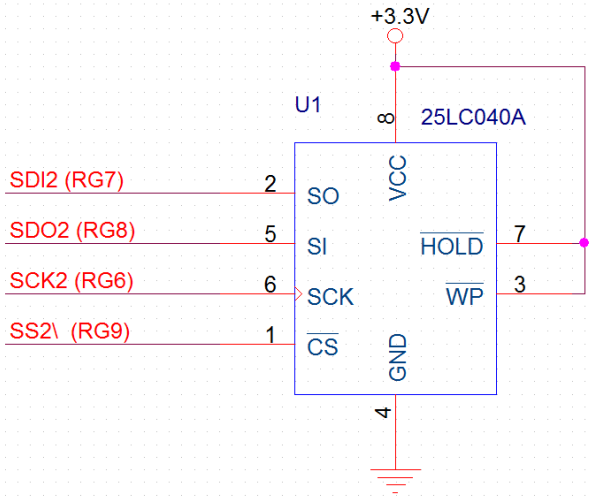
O registo de **STATUS** pode ser escrito usando um protocolo em que são enviados dois bytes em sequência: um byte com o comando **WRSR** e outro byte com o valor a escrever (ver página 11 do manual da EEPROM).

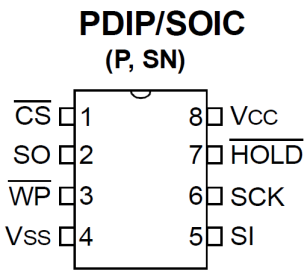
O fabricante disponibiliza também uma sequência específica para a desproteção da escrita (ativação do bit **WEL** do registo de **STATUS**). Esta sequência envolve apenas a transmissão de 1 byte correspondente ao código do comando (**WREN**, ver Figura 27).

Trabalho a realizar

Parte I

1. Monte, na placa branca, a memória EEPROM 25LC040A, de acordo com a figura seguinte. Os sinais **SDI2** (**RG7**), **SDO2** (**RG8**), **SCK2** (**RG6**) e **SS2** (**RG9**) correspondem à ligação ao módulo SPI número 2 do PIC32.





2. Escreva uma função para configurar o gerador de *baudrate*:

```
void spi2_setClock(unsigned int clock_freq)
{
   // Write SPI2BRG register(see introduction for details)
}
```

3. Implemente a função para inicializar o módulo SPI (consulte o manual do SPI).

```
void spi2_init(void)
{
   volatile char trash;
   // Disable SPI2 module
   // Configure clock idle state as logic level 0
   // Configure the clock active transition: from active
   //   state to idle state
   // Configure SPI data input sample phase bit (middle of data
   //   output time)
   // Configure word length (8 bits)
   // Enable Enhanced buffer mode (this allows the usage of FIFOs RX,TX)
   // Enable slave select support (Master Mode Slave Select)
   // Enable master mode
   // Clear RX FIFO:
   while(SPI2STATbits.SPIRBE == 0)    // while RX FIFO not empty read
      trash = SPI2BUF;                // FIFO and discard read character
   // Clear overflow error flag
   // Enable SPI2 module
}
```

4. As funções que se vão implementar de seguida são dependentes do modo de funcionamento da EEPROM já explicado na introdução deste trabalho prático. Vamos começar por implementar a função que permite a leitura do registo de **STATUS**, uma vez que essa função é necessária para todas as restantes:

```
char eeprom_readStatus(void)
{
   volatile char trash;

   // Clear RX FIFO
   // Clear overflow error flag bit
   SPI2BUF = RDSR;   // Send RDSR command
   SPI2BUF = 0;      // Send anything so that EEPROM clocks data into SO
   while(SPI2STATbits.SPIBUSY);  // wait while SPI module is working
   trash = SPI2BUF;  // First char received is garbage (received while
                     //   sending command)
   return SPI2BUF;   // Second received character is the STATUS value
}
```

   De modo a tornar o programa mais legível, deverá definir os códigos dos comandos que estão definidos no manual do fabricante. Por exemplo:

```
#define RDSR   0x05
#define WRITE  0x02
```

5. Teste as funções que já escreveu. Para isso escreva o programa principal que chame as funções de inicialização e que, em ciclo infinito, leia o registo de **STATUS** e imprima o seu valor usando *system calls* (apenas os 4 bits menos significativos do byte recebido têm informação útil). Se as funções funcionarem adequadamente, o valor recebido nos 4 bits menos significativos deverá ser 0 (**BP1**, **BP0**, **WEL** e **WIP** todos a 0),

```
void main(void)
{
   spi2_init();
   spi2_setClock(EEPROM_CLOCK);
   for(;;)
   {
      // Call "eeprom_readStatus() function
      // Print read value
   }
}
```

   O sinal de relógio para a EEPROM pode, de acordo com o fabricante, ter uma frequência máxima de 5 MHz. Não é, contudo, aconselhável a utilização de uma frequência tão elevada numa montagem em placa branca, pelo que se recomenda a utilização de uma frequência mais baixa, por exemplo 500 KHz (a frequência mais pequena que poderá utilizar é 20 kHz. Porquê?).

6. Escreva agora a função para implementar a sequência simplificada de escrita no registo **STATUS** (apenas para os comandos **WREN** e **WRDIS**):

```
void eeprom_writeStatusCommand(char command)
{
   while( eeprom_readStatus() & 0x01 );     // Wait while WIP is true
                                            //   (write in progress)
   // Copy "command" value to SPI2BUF  (TX FIFO)
   // Wait while SPI module is working (SPIBUSY set)
}
```

7. Teste a função anterior acrescentando ao programa que escreveu no exercício 5 a chamada da função **eeprom_writeStatusCommand()** com o argumento **WREN**. O valor

retornado pela função de leitura do registo de **STATUS** deve agora ser 2 (isto é, a proteção de escrita está desativada).

8. Escreva a função para escrita de um byte numa posição de memória. A função deve ter como parâmetros de entrada o endereço de escrita e o valor a escrever:

```
void eeprom_writeData(int address, char value)
{
   // Apply a mask to limit address to 9 bits
   // Read STATUS and wait while WIP is true (write in progress)
   // Enable write operations (activate WEL bit in STATUS register, using
         eeprom_writeStatusCommand() function )
   // Copy WRITE command and A8 address bit to the TX FIFO:
   SPI2BUF = WRITE | ((address & 0x100) >> 5);
   // Copy address (8 LSBits) to the TX FIFO
   // Copy "value" to the TX FIFO
   // Wait while SPI module is working (SPIBUSY)
}
```

9. Escreva, finalmente, a função para leitura de um byte de uma posição de memória. A função deve ter como parâmetro de entrada o endereço e deve retornar o valor lido:

```
char eeprom_readData(int address)
{
   volatile char trash;
   // Clear RX FIFO
   // Clear overflow error flag bit
   // Apply a mask to limit address to 9 bits
   // Read STATUS and wait while WIP is true (write in progress)
   // Copy READ command and A8 address bit to the TX FIFO
   // Copy address (8 LSBits) to the TX FIFO
   // Copy any value (e.g. 0x00) to the TX FIFO
   // Wait while SPI module is working (SPIBUSY)
   // Read and discard 2 characters from RX FIFO (use "trash" variable)
   // Read RX FIFO and return the corresponding value
}
```

10. Para o teste das funções que escreveu nos exercícios anteriores escreva a função **`main()`**, de modo a realizar, em ciclo infinito, as seguintes operações (utilize system calls para a interação com o utilizador):
    - Lê um caracter
    - Se for 'R' (read) lê um endereço (**`addr`**), e imprime o valor lido da memória.
    - Se for 'W' (write) lê um endereço e um valor (**`addr`**, **`val`**), e escreve na EEPROM no endereço **`addr`** o valor **`val`**.

```
void main(void)
{
   for(;;)
   {
      // Read character
      // If character is 'R' then ...
      (...)
   }
}
```

11. Deslige a alimentação durante, pelo menos, 30 segundos, e repita o exercício anterior efetuando apenas leituras dos endereços de memória em que escreveu.
12. Altere a função que implementou no exercício anterior de modo a que, após a leitura do endereço e do valor, o programa escreva sucessivamente nas 16 posições de memória seguintes o valor anterior incrementado de 1. De seguida o programa deve ler os 16 valores da EEPROM e imprimi-los em hexadecimal (formato: "endereço inicial: valor valor ...).

Parte II

1. O objetivo deste exercício é observar os sinais da interface SPI com o osciloscópio. Para isso retome o código que escreveu no exercício 10 e coloque a chamada à função de leitura da EEPROM em ciclo infinito. Ligue as duas pontas de prova do osciloscópio, uma à linha **CS** e outra à linha **SI** da EEPROM e selecione como entrada de *trigger* do osciloscópio o canal que ligou à linha **CS**. Observe o sinal na linha **SI** e identifique os valores transferidos. Repita o procedimento colocando a ponta de prova na linha **SO** da EEPROM.
2. À semelhança do que já fez no trabalho prático anterior, pretende-se agora organizar o código produzido de forma a que ele possa facilmente ser integrado em outras aplicações. Para isso escreva o ficheiro "**eeprom.h**" com os protótipos de todas as funções e os símbolos públicos. Construa também o ficheiro "**eeprom.c**" com o código das funções.

```
// eeprom.h
#ifndef EEPROM_H
#define EEPROM_H
// Declare symbols here (READ, WRITE, ...)
(...)
// Declare function prototypes here
(...)
#endif
```

3. Retome o código que escreveu no último exercício do trabalho prático n.º 10. Faça as alterações que permitam o registo de temperatura em EEPROM, bem como a sua visualização, de acordo com a seguinte especificação:
   1) os valores de temperatura deverão ser armazenados a partir do endereço **0x002** da EEPROM; a posição de memória **0x000** deverá conter o número de valores de temperatura armazenados;
   2) quando for recebido da linha série o caracter 'R' (*reset*) o sistema deve colocar a zero o número de temperaturas armazenado na memória (endereço **0x000** da EEPROM);
   3) quando for recebido da linha série o caracter 'L' (*log*) o sistema deve iniciar o registo, na EEPROM (a partir do endereço **0x002**), do valor instantâneo da temperatura; deverão ser armazenadas 4 amostras por minuto (i.e. uma amostra a cada 15s), até um máximo de 64; a posição de memória **0x000** deve ser incrementada a cada nova escrita na EEPROM (ou seja, a posição de memória **0x000** deverá conter o número de valores de temperatura válidos armazenados em memória);
   4) quando for recebido o caracter 'S' (*show*) o sistema deve parar o registo de temperaturas e enviar para a porta série os valores de temperatura entretanto armazenados;

   Note que o comando 'R' terá que ser efetuado, pelo menos uma vez, no início, para se colocar a posição de memória **0x000** a zero.
4. Altere o código do exercício anterior de modo a efetuar o registo contínuo de temperatura, isto é, sem a limitação dos 64 valores. Para isso deverá implementar na memória um *buffer* circular de 64 posições, a partir do endereço **0x002**. A posição de memória **0x000** deverá conter o número de valores armazenados (com um máximo de 64) e a posição **0x001** deverá conter o endereço onde a próxima escrita deverá ser efetuada.

Elementos de apoio

- Slides das aulas teóricas.
- 25LC040A - 4K SPI Bus Serial EEPROM (disponível no site da disciplina).
- PIC32 Family Reference Manual, Section 23 – SPI.