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.

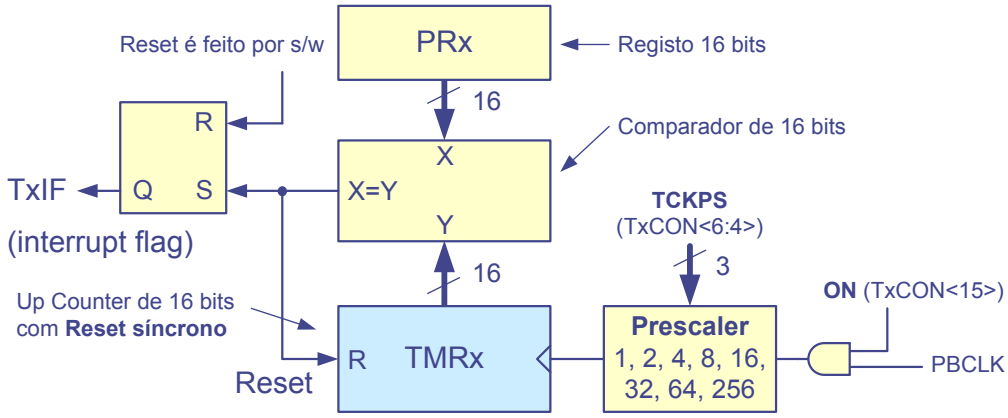

Figura 12. Diagrama de blocos simplificado de um timer tipo B.

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).

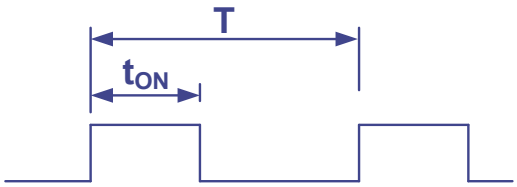

Figura 13. Exemplo de sinal rectangular com um período T e um tempo a 1 t_{ON}.

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 = (tON / 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.

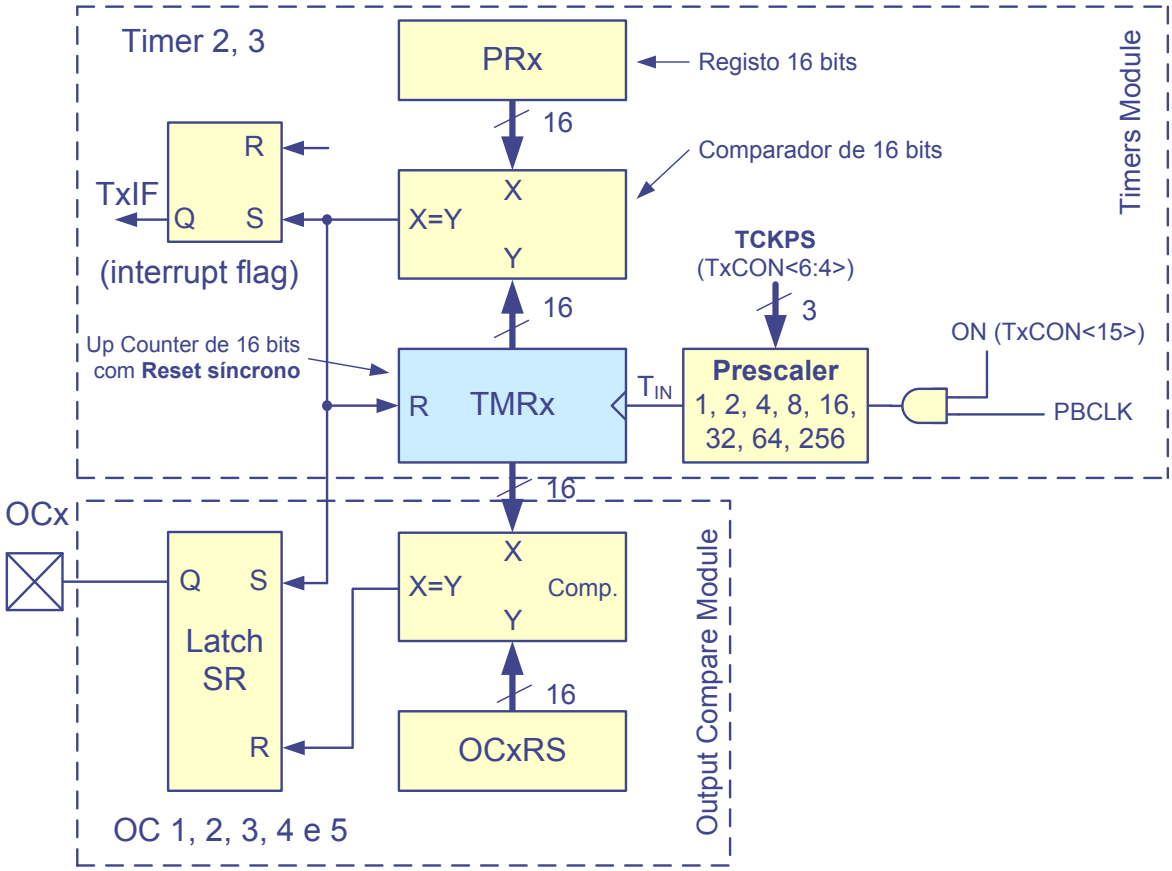

Figura 14. Diagrama de blocos do sistema de geração de sinais PWM.

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:

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

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.