

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 recorrendo ainda a um módulo designado pelo fabricante por *Output Compare Module*.

No PIC32MX795F512H (versão usada na 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 de tipo B reside no módulo *prescaler* (pré-divisor) que apenas permite, no de tipo A, a divisão por 1, 8, 64 ou 256. Nos de tipo B a constante de divisão pode ser 1, 2, 4, 8, 16, 32, 64 ou 256. Os *timers* do tipo B podem ser agrupados dois a dois implementando, desse modo, um *timer* de 32 bits. A Figura 1 apresenta o diagrama de blocos simplificado de um *timer* tipo B do PIC32.

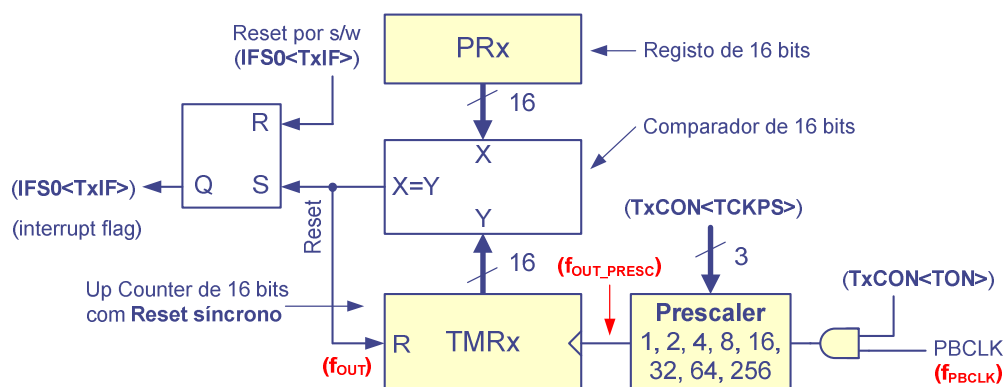


Figura 1. Diagrama de blocos simplificado de um *timer* tipo B.

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, $f_{PBCLK} = 20 \text{ MHz}$ (**FREQ/2**, ou **PBCLK** em C).

Cálculo das constantes para geração de um evento periódico

O módulo de pré-divisão (*prescaler*) faz uma divisão da frequência f_{PBCLK} por uma constante configurável nos 3 bits **<TCKPS>** do registo **TxCON**¹ (designada mais à frente por **K_{PRESCALER}**), para os *timers* T2 a T5, ou nos 2 bits **TCKPS** do registo **T1CON**, para o *timer* T1. Por exemplo, no *timer* tipo A, se **<TCKPS>** for configurado com o valor 3, a que corresponde uma constante de divisão de 256, o valor de f_{OUT_PRESC} obtido é:

$$f_{OUT_PRESC} = \frac{f_{PBCLK}}{256}$$

¹ Para informação completa sobre o modelo de programação, deve ser consultado o manual do fabricante "PIC32 Family Reference Manual, Section 14 – Timers".

Conhecida a frequência do sinal à saída do *prescaler*, pode determinar-se a frequência do sinal gerado pelo *timer*, do seguinte modo:

$$f_{OUT} = \frac{f_{OUT_PRESC}}{PRx + 1}$$

em que **PRx** é o valor da constante de 16 bits armazenada num dos registos **PR1** a **PR5** (*timers* T1 a T5).

Exemplo: determinar o valor de **PR2** e da constante de divisão do *prescaler* de modo a que o *timer* T2 gere eventos de "fim de contagem" a uma frequência de 10 Hz (i.e. a cada 100 ms).

Se o *prescaler* for configurado com o valor 1, então $f_{OUT_PRESC} = f_{PBCLK} = 20 \text{ MHz}$ e **PR2** fica:

$$PR2 = \left(\frac{20 \times 10^6}{10} \right) - 1 \quad (\text{em C, } PR2 = PBCLK / 10 - 1;)$$

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* para baixar a frequência do sinal à entrada do contador do *timer*, de modo a tornar possível a divisão usando uma constante de 16 bits.

$$f_{OUT} = \frac{(f_{PBCLK} / K_{PRESCALER})}{(PR2 + 1)}$$

Usando para **PR2** o valor máximo possível (**65535**), podemos determinar o valor mínimo para a constante de divisão do *prescaler* como:

$$K_{PRESCALER} = \left\lceil \frac{f_{PBCLK}}{((65535 + 1) \times f_{OUT})} \right\rceil = [30.51] = 31$$

Se, por exemplo, se usar uma constante de divisão de 32 (os valores possíveis seriam 32, 64 ou 256), $f_{OUT_PRESC} = 20 \text{ MHz} / 32 = 625 \text{ KHz}$. Refazendo o cálculo para o valor de **PR2** obtém-se:

$$PR2 = \left(\frac{625 \times 10^3}{10} \right) - 1 = 62499$$

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

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

Configuração do *timer*

A programação dos *timers* envolve: i) configuração da constante de divisão do *prescaler* (registo **TxCON**, bits **<TCKPS>**), ii) configuração da constante de divisão **PRx**, iii) ativação do *timer* (registo **TxCON**, bit **<TON>**). A sequência para a configuração do *timer* T2 com os parâmetros do exemplo anterior é:

```
T2CONbits.TCKPS = 5; // 1:32 prescaler (i.e. fout_presc = 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**, bits **<TxIP>**), *enable*

das interrupções geradas pelo *timer* pretendido (registo **IEC0**, bits **<TxIE>**) e *reset* inicial do bit **<TxIF>** (registo **IFS0**)². Para o *timer* T2, a sequência de comandos que configura o sistema de interrupções fica então:

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

Geração de um sinal PWM

PWM (*Pulse Width Modulation*, ou modulação por largura de pulso) 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 retangulares, como o apresentado na Figura 2, em que, mantendo o período T , se pode alterar dinamicamente a duração a 1, t_{ON} , do sinal.

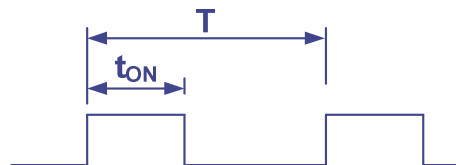


Figura 2. Exemplo de sinal retangular com um período T e um tempo a 1 t_{ON} .

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:

$$\text{Duty-cycle} = \frac{t_{ON}}{T} \times 100[\%]$$

No PIC32 a geração de sinais PWM é feita usando os *timers* T2 ou T3 e o *Output Compare Module* (OC). A Figura 3 apresenta o diagrama de blocos desse sistema, onde se evidencia a interligação entre o módulo correspondente aos *timers* T2 e T3 e o módulo OC.

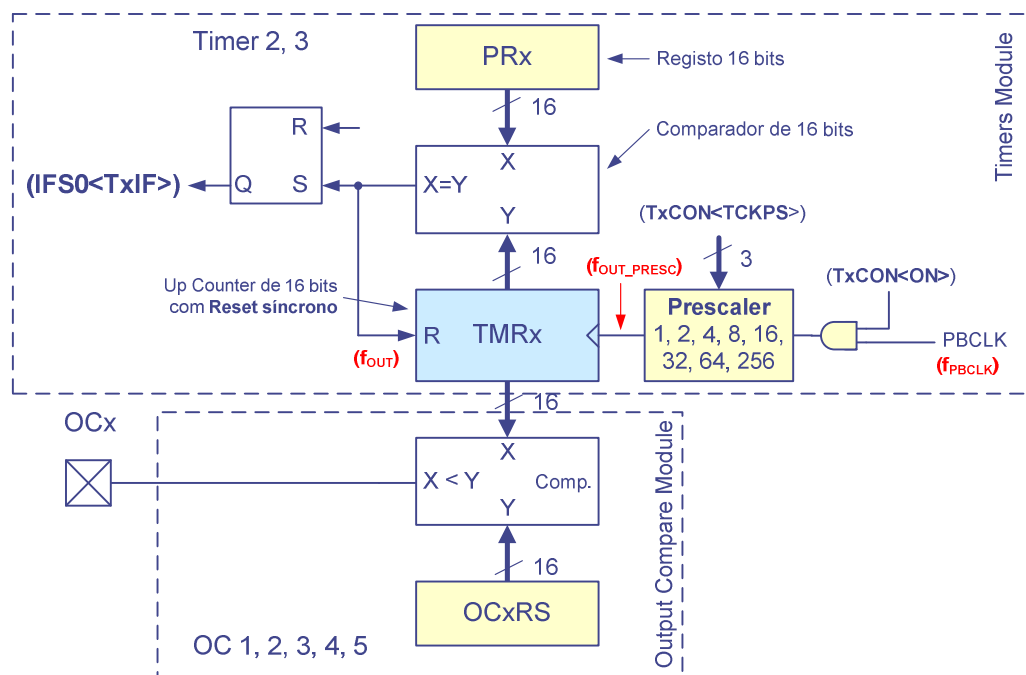


Figura 3. Diagrama de blocos do sistema de geração de sinais PWM.

² 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. 74 a 76 (ambos disponíveis no site da UC).

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**:

$$t_{ON} = 0.2 \times TPWM = 0.2 \times \left(\frac{1}{10}\right) = 20ms$$

$$f_{OUT_PRESC} = 625KHz, \quad T_{OUT_PRESC} = \frac{1}{62500} = 1.6\mu s$$

Então **OC1RS** deverá ser configurado com:

$$OC1RS = \frac{20 \times 10^{-3}}{1.6 \times 10^{-6}} = 12500$$

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

$$OC1RS = \frac{((PR2 + 1) * duty-cycle)}{100} = \frac{((62499 + 1) * 20)}{100} = 12500$$

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:

$$\text{Resolução} = \log_2 \left(\frac{T_{PWM}}{T_{OUT_PRESC}} \right) = \log_2 \left(\frac{f_{OUT_PRESC}}{f_{OUT}} \right)$$

Para as frequências do exemplo anterior a resolução é então: $\log_2(625000/10) = 15 \text{ bits}$

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 Fout_presc = 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 (ou seja, 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** (**IFS0<T3IF>**) e envie para o ecrã o carácter ' . ' sempre que esse bit fique ativo:

```
void main(void)
{
    // Configure Timer T3 (2 Hz with 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 with interrupts enabled
    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 (como poderá facilmente verificar não é possível obter diretamente, através do timer, a frequência de 1 Hz; uma solução será chamar o *system call* a cada 2 interrupções).
4. Pretende-se neste exercício a configuração do sistema de interrupções e dos *timers* 1 e 3: o *timer* 1 a gerar interrupções à frequência de 2 Hz e o *timer* 3 a gerar interrupções à frequência de 10 Hz.
 - a) Determine as constantes relevantes para que o *timer* T1 (tipo A) gere eventos de interrupção a cada 500 ms (2 Hz) e o *timer* T3 (tipo B) gere eventos de interrupção a cada 100 ms (10 Hz).
 - b) Escreva o programa principal com todas as configurações necessárias e as Rotinas de Serviço à Interrupção dos *timers* 1 e 3. Nessas rotinas deve apenas imprimir o carácter '1' na RSI do *timer* 1 e o carácter '3' na RSI do *timer* 3.

```
void main(void)
{
    // Configure Timers T1 and T3 with interrupts enabled)
    // Reset T1IF and T3IF flags
    EnableInterrupts();           // Global Interrupt Enable
    while(1);
}
```

```

void _int_(VECTOR_TIMER1) isr_T1(void)
{
    // print character '1'
    // Reset T1IF flag
}

void _int_(VECTOR_TIMER3) isr_T3(void)
{
    // print character '3'
    // Reset T3IF flag
}

```

Verifique o correto funcionamento do sistema, observando no ecrã do PC a sequência de números impressa. Altere a frequência das interrupções do *timer* 3 para 20 Hz e verifique novamente o funcionamento do sistema.

5. 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. O sistema de visualização funcionava com uma frequência de refrescamento de 100 Hz (10 ms). Ainda nesse exercício, os tempos relevantes (10 ms e 250 ms) 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:

- *timer* T1: determina a frequência de amostragem, i.e., o ritmo de leitura da entrada analógica;
 - *timer* T3: determina a frequência de refrescamento do sistema de visualização.
- a) Determine 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).
- b) Escreva o programa principal, onde, no essencial, se faz a configuração de todos os dispositivos em utilização e se ativam globalmente as interrupções.

```

volatile int voltage = 0;    // Global variable

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

```

- c) Escreva a rotina de serviço à interrupção do *timer* T1, onde deve ser dada a ordem de início de conversão à ADC.

```

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

```

- d) Escreva a rotina de serviço à interrupção do *timer* T3, onde deve ser feito o envio para o sistema de visualização do valor de tensão calculado pela rotina de serviço à interrupção da ADC.

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

- e) Integre no conjunto a rotina de serviço à interrupção da ADC (já implementada anteriormente).

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

Nesta fase o sistema deverá estar a funcionar integralmente por interrupção, convertendo o valor da tensão analógica presente na entrada **AN4** e a mostrar o respetivo valor nos dois *displays*.

6. Pretende-se agora dotar o sistema de uma funcionalidade adicional que permita a paragem temporária da conversão, ficando os *displays* a mostrar o último valor de tensão medido (*freeze*). Para isso configure os portos **RB1** e **RB0** como entrada e faça as alterações ao código que permitam parar a conversão quando o valor lido desses dois portos tiver a combinação binária "01" (**RB1=0; RB0=1**). Sugestão: controle o bit de *enable/disable* das interrupções do *timer* T1 (*timer* que controla o ritmo de conversão da ADC).

Parte II

- 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.
- Escreva uma função que permita (para a frequência de 100 Hz) configurar o módulo **oc1** para gerar qualquer valor de *duty-cycle* 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"
}
```

- 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 (ligado ao porto **RD0** da placa DETPIC32) 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.
- Pretende-se agora integrar o controlo do brilho do LED D1 no programa que escreveu no ponto 5 da parte 1. Para isso, os bits **RB1** e **RB0** 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 (LED ON com o brilho no máximo)
1X - brilho do LED depende do valor da tensão medido pelo sistema
```

Para fazer depender o *duty-cycle* do valor da tensão medido pelo sistema (disponível na variável global "voltage") poderá fazer `dutyCycle = 3 * voltage`, e obterá valores entre 0 e 99.

```
volatile int voltage;

void main(void)
{
    int dutyCycle;
    configureAll();
    EnableInterrupts(); // Global Interrupt Enable
    while(1)
    {
        // Read RB1, RB0 to the variable "portVal"
        switch(portVal)
        {
            case 0: // Measure input voltage
                // Enable T1 interrupts
                setPWM(0); // LED OFF
                break;
            case 1: // Freeze
                // Disable T1 interrupts
                setPWM(100); // LED ON (maximum bright)
                break;
            default: // LED brightness control
                // Enable T1 interrupts
                dutyCycle = voltage * 3;
                setPWM(dutyCycle);
                break;
        }
    }
}
```

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 Datasheet, Pág. 74 a 76.