UNIVERSIDADE
FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

CURSO DE GRADUAÇÃO EM **ENGENHARIA DE COMPUTAÇÃO**
SÉTIMO SEMESTRE

QXD0143 - MICROCONTROLADORES

Relatório 04: Contando Tempo com os Temporizadores (*Timers*)
LPTMR, PIT e TPM

QUIXADÁ - CE
2022

471047 — PEDRO HENRIQUE **MAGALHÃES BOTELHO**

495519 — RAFAEL DA **SILVA GONÇALVES**

Relatório 04: Contando Tempo com os Temporizadores (*Timers*) LPTMR, PIT e TPM

Orientador: Prof. Dr. Thiago Werlley Bandeira da Silva

Quarto relatório escrito para a disciplina de Microcontroladores, no curso de graduação em Engenharia de Computação, pela Universidade Federal do Ceará (UFC), campus em Quixadá.

QUIXADÁ - CE
2022

Conteúdo

Lista de Figuras

1 Introdução

Esse é o quarto relatório da disciplina de **Microcontroladores**, ministrada pelo professor Thiago W. Bandeira. Nesse relatório é discutido sobre a utilização de temporizadores, do inglês *timers*, para a contagem de tempo de forma precisa em microcontroladores ARM. Nesse relatório utilizamos três módulos de *timers* próprios de microcontroladores Kinetis: **LPTMR**, **PIT** e **TPM**.

Nas práticas realizadas em laboratório foi utilizada a placa de desenvolvimento **Freedom FRDM-KL25Z**, da Kinetis/NXP, com o microcontrolador **MKL25Z128VLK4**.

Este documento contém informações e imagens retiradas do manual de referência do microcontrolador usado, como mostra as figura 01:

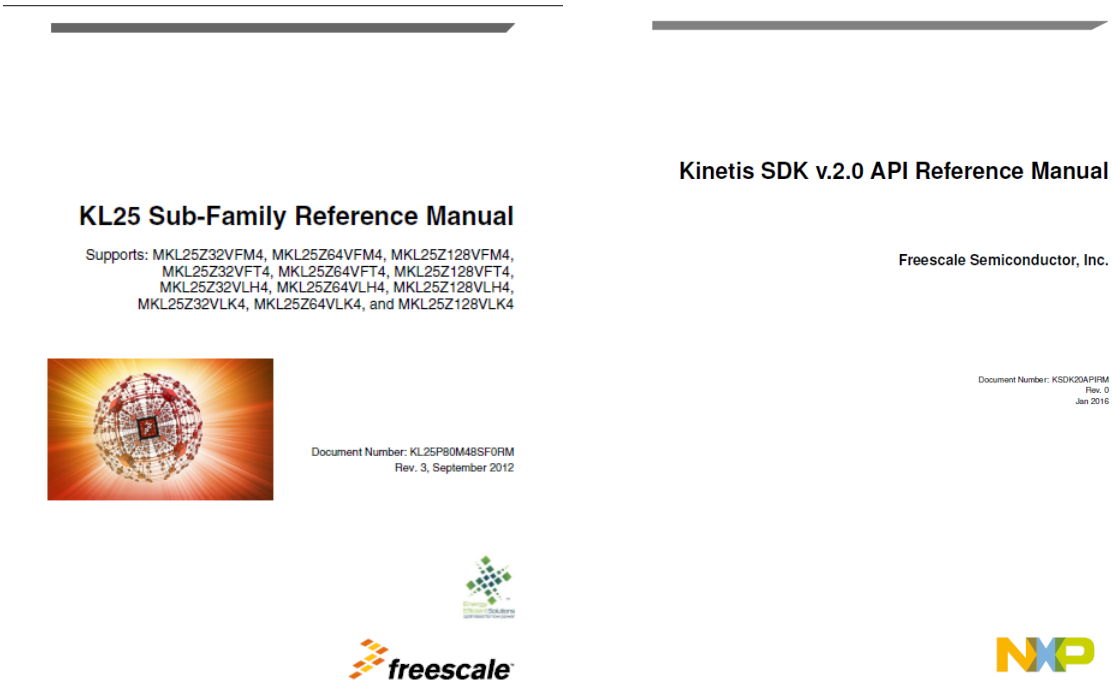

Figura 1: Capas dos Manuais de Referência do MKL25Z128VLK4 e do SDK.

Também foi utilizado o manual de referência do *Software Development Kit* da Kinetis, o **SDK**, como mostra a figura 01. O SDK é uma API que fornece diversas ferramentas para a programação de microcontroladores da Kinetis/NXP.

Os projetos da disciplina estão disponíveis em um repositório no Github, em [1], onde estão os relatórios, códigos-fonte e esquemáticos (os projetos completos encontram-se comprimidos em arquivos .zip). O link também se encontra nas referências.

2 Temporizadores

Até agora estávamos usando uma função específica para contar algum tempo, uma função de *delay*, em que passávamos o tempo solicitado em milissegundos. A função, então, iria ocupar a CPU durante o tempo solitiado, com um laço, como mostra o código abaixo:

```
1 void delay_ms(int ms) {
2   for(int i = 0; i < ms; i++) {
3     for(int j = 0; j < 7000; j++);
4   }
5 }
```

O grande problema disso era que, durante esse tempo a CPU ficava ocupada, já que essa função se utilizava da quantidade de ciclos de *clock* que eram empregados para contar o tempo. As vezes é útil bloquear a CPU, para que o programa só continue após um certo tempo, porém, geralmente, não é interessante bloquear a CPU toda as vezes que for necessário esperar algum tempo.

Para contornar esse problema utilizamos um hardware externo à CPU que irá contar o tempo de forma paralela à CPU, os **temporizadores**, ou *timers*. Dessa forma, a CPU pode executar tarefas enquanto um hardware temporizador conta o tempo. Quando o tempo requisitado for atingido a CPU é avisada por meio de um sinal IRQ, mandado pelo temporizador.

2.1 Funcionalidade Básica

Podemos imaginar o temporizador (por vezes também chamado de contador) como um hardware genérico com diversas funções, tais como medir um tempo através de um sinal de clock de referência ou contar eventos externos. Iremos focar na função de contagem de tempo.

O hardware genérico do temporizador/contador (comum à todos os periféricos temporizadores dos microcontroladores) é mostrado na figura 02, sendo então muito semelhante ao **circuito digital contador**. No caso do periférico temporizador, um **registrador contador** mantém a contagem, que é incrementada conforme o sinal de contagem, que é o sinal proveniente de uma fonte de *clock* dividido por um *prescaler*, um circuito que divide a frequência do sinal.

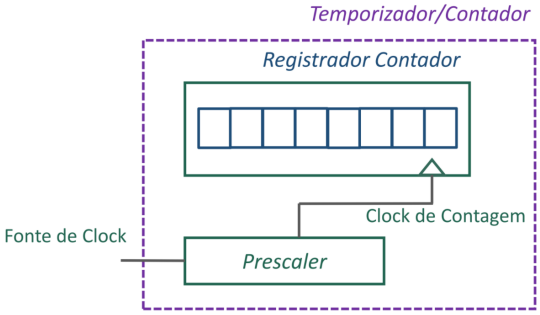

Figura 2: Modelo de *hardware* genérico de um temporizador.

A cada borda de subida do *clock* de contagem o valor mantido no registrador contador incrementada em uma unidade. Dessa forma, temos uma contagem que se baseia somente em um sinal de referência.

Resumindo: o temporizador realiza a contagem de tempo sem usar o tempo da CPU, sendo um circuito a parte. Quando o instante de tempo requisitado for atingido uma interrupção é gerada, indicando à CPU a passagem do tempo. Veremos que com o temporizador realizaremos contagens com um tempo pré-definido cada, e, a partir do tempo de cada contagem, definiremos o tempo total a ser contado, definido a quantidade de contagens.

2.2 Etapas do *Timer*

Podemos dividir o funcionamento do temporizador, ou *timer*, em alguns estágios, como mostra a figura 03. Explicitaremos cada estágio, e, ao final, daremos um exemplo de como cálcular o tempo e a quantidade de contagens.

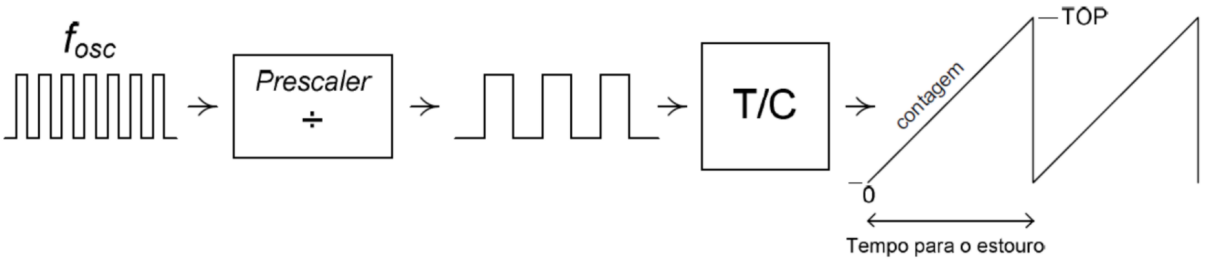

Figura 3: Etapas do periférico Temporizador.

No primeiro estágio, uma fonte de *clock* emite um sinal de referência ao periférico. Definimos via *software* qual a fonte de clock para o temporizador. Um *hardware* específico chamado *prescaler* irá dividir esse *clock* em um sinal de menor frequência, emitindo a frequência de contagem, que será usada pelo contador para realizar uma contagem. Essa contagem inicia em zero e vai até **TOP**, que é o valor máximo da contagem, definido via *software*.

Quando a contagem chega em TOP (o valor da contagem é igual a TOP) a contagem é reiniciada (o registrador irá para zero e a contagem começará novamente). O ato do registrador ir do valor TOP para zero caracteriza o estouro de contagem, o **overflow**. O tempo entre *overflows* pode ser visto como o tempo de estouro de 0 a TOP, ou seja, o tempo que queremos contar.

2.3 Cálculo de Tempo do *Timer*

Vamos entender o cálculo do tempo usando um exemplo. Temos:

- Temporizador de 8-bits (registrador de 8-bits).

- Fonte de *Clock* de 16 MHz.
- *Prescaler* de 64 (divide o *clock* por 64).

Devemos então ser capazes de responder **três perguntas** para podermos definir precisamente o tempo que o contador irá contar:

2.3.1 Qual o módulo de contagem?

O módulo de contagem é o número de contagens necessário para atingir o estouro de contagem, ou seja, o número de contagens feitas de 0 até o *overflow*.

Além do registrador contador temos um registrador de mesmo tamanho que mantém o módulo, que é igual a TOP - 1. Podemos dizer que o módulo (MOD) é o número de contagens máximo de um temporizador se utilizarmos a capacidade total do registrador MOD. Nesse caso, temos que o módulo máximo é:

$$\text{Módulo (MOD)} = 2^n, \text{ onde n é a quantidade de bits do registrador.}$$

Logo:

$$\text{Módulo (MOD)} = 2^8 = 256 \text{ contagens.}$$

Temos então que com um registrador de 8-bits o número máximo de contagens de 0 até o *overflow* é 256: de 0 a 255 (valor máximo).

2.3.2 Qual o período de uma contagem?

O período da contagem é o tempo necessário para realizar uma contagem, ou seja, o tempo necessário para incrementar em um o registrador contador, como mostra a figura 04.

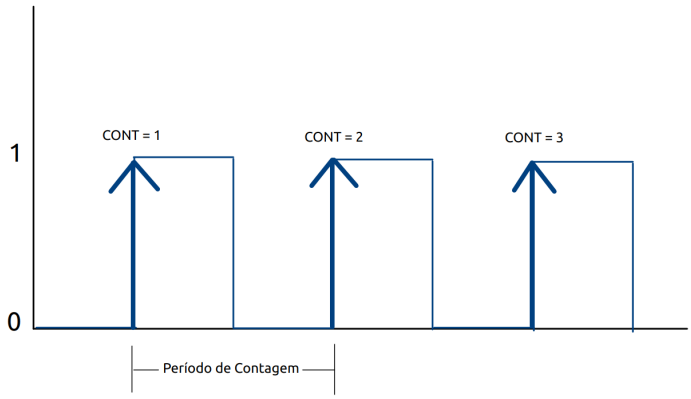

Figura 4: Período de contagem do temporizador.

Se olharmos por um ponto de vista físico, o período, em segundos (s), é o tempo para que se complete um ciclo (de *clock*, no caso), sendo inverso da frequência, em Hertz (Hz), que indica a quantidade de ciclos em um segundo. Quanto maior o período, menor a frequência e vice versa. Temos então a seguinte relação:

$$T = \frac{1}{f}, \text{ onde } f \text{ é a frequência de contagem, que é } f = \frac{f_{source}}{ps}.$$

Temos que f_{source} indica a frequência do sinal proveniente da fonte de *clock*, e ps indica o valor do *prescaler*, que irá dividir o *clock* original. Substituindo os valores, temos:

$$f = \frac{f_{source}}{ps} = \frac{16 \times 10^6}{64} = 250 \times 10^3 \text{ Hz ou } 250 \text{ kHz}$$

$$T = \frac{1}{250 \times 10^3} = 4\mu s$$

Portanto, o período de contagem equivale a 4μ. Ou seja, cada contagem levará $4\mu s$, como mostra a figura 05.

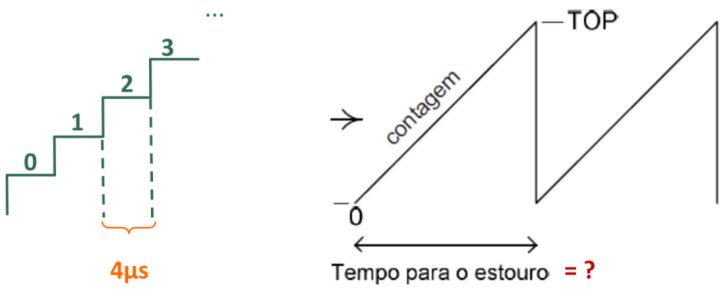

Figura 5: Período de contagem do exemplo é $4\mu s$.

2.3.3 Qual o período para o estouro da contagem?

Devemos agora descobrir que tempo iremos contar com um módulo de 256 e com um período de contagem de $4\mu s$. O período para o estouro é, então, o tempo desde o começo da contagem até reiniciar a contagem.

Se o registrador pode realizar 256 contagens antes do seu *overflow* aritmético ($255 + 1 = 0$, em 8-bits), e cada contagem leva $4\mu s$, temos que:

$$256 \times 4 \times 10^{-6} = 1,024 \times 10^{-3}, \text{ ou } 1,024ms$$

Ou seja, o tempo total da contagem, desde o início até o estouro é de $1,024ms$. Podemos generalizar então para:

$$\text{Quantidade de Contagens} \times \text{Período de Contagem} = \text{Tempo até o } \textit{Overflow}$$

2.4 Interrupção por *Overflow* de Contagem

Como já discutido antes, quando a contagem chega no valor do módulo e é incrementada novamente ocorre o estouro de contagem, ou *overflow*, fazendo o registrador contador ir para zero e o temporizador lançar um sinal de interrupção (IRQ) à CPU para indicar que passou-se o tempo determinado. A CPU então irá tratar essa interrupção, invocando a ISR relativa ao periférico (ou a um módulo específico do periférico, dependendo do temporizador). Dessa forma estabelecemos uma periodicidade, já que o temporizador lançará a IRQ de tempos em tempos, baseado no tempo configurado.

Quando vamos definir um tempo para o temporizador contar devemos configurar o *clock* para o periférico, o *prescaler* e o valor do módulo tendo em mente o cálculo que usamos para definir o tempo. Portanto, os valores utilizados podem facilitar e nos permitem atingir os valores esperados.

Vamos apresentar um roteiro para definir o módulo para realizar as contagens. Para isso, vamos usar como exemplo o temporizador *Timer/PWM Module* (TPM), um dos cinco temporizadores da placa KL25Z. A título de curiosidade, os outros quatro são: *Low-Power Timer* (LPTMR), *Periodic Interrupt Timer* (PIT), SysTick e *Real Time Clock* (RTC), sendo esses dois últimos comuns a todos os processadores ARM.

O primeiro passo é definir a frequência do temporizador. Vamos usar como exemplo o *clock* padrão da KL25Z, de 20,972 MHz. Depois devemos definir o valor do *prescaler*, que irá dividir essa frequência. O valor do *prescaler* deve ser definido tendo em mente a *frequência* que queremos obter, de forma que facilite os cálculos ou que obtenhamos um tempo maior. Se quisermos obter o menos clock possível nos escolhemos o maior valor para o *prescaler*, no caso do TPM, 128.

Vamos definir então a frequência de contagem e o período de cada contagem:

- Frequência de Contagem: $f = \frac{20{,}972 \times 10^6}{128} = 163,84 \times 10^3$ Hz ou $163,84$ kHz.
- Período de cada Contagem: $T = \frac{1}{163{,}84 \times 10^3} = 6,103\mu$s.

Se quisermos definir um tempo de 300 ms, quantas contagens seriam necessárias, com essas configurações? Temos a seguinte relação:

$$\text{Quantidade de Contagens} = \text{Tempo até o } \textit{Overflow} \ / \ \text{Período de Contagem}$$

$$\text{Quantidade de Contagens} = 300 \times 10^{-3} \ / \ 6,103 \times 10^{-6} \approx 491456 \text{ contagens.}$$

Ou seja, demos então que se a contagem for realizada de 0 até 491455 teremos um tempo de 300ms. Devemos então configurar o registrador de módulo com o valor 491455, referente ao TOP.

3 Low-Power Timer (LPTMR)

O LPTMR tem apenas um módulo e tem algumas opções de fontes de *clock* configuráveis, como mostra a figura 06. Dentre as opções temos o *clock* de 1 kHz, provindo do *Low Power Oscillator*, o oscilador interno **LPO**, e o clock do oscilador externo, o *Oscillator External Reference Clock*, o **OSCERCLK**, de 32,768 kHz.

O LPTMR conta com um registrador de comparação de 16-bit, podendo realizar até 65536 contagens em cada período de tempo. O LPTMR conta, ainda, com a opção de "desviar" do *prescaler*, utilizando a mesma frequência de *clock* que chega ao periférico (feito atingido por outros temporizadores com um *prescaler* de ÷1). O periférico, ainda, pode ser configurado para contador de tempo ou contador de pulsos.

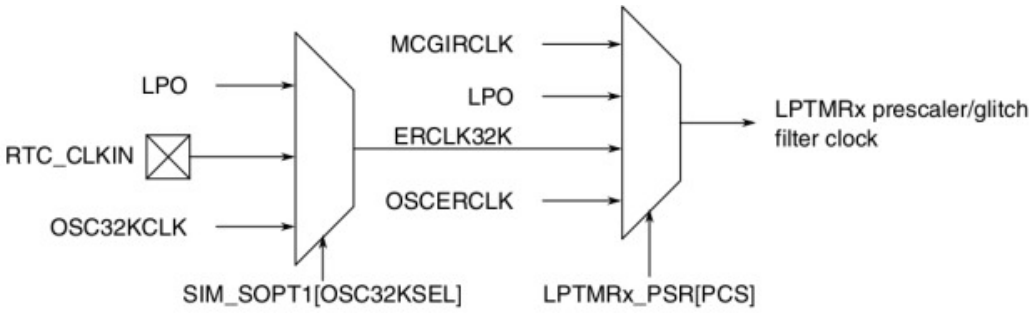

Figura 6: Modelo Genérico do modulo LPTMR.

3.1 Funcionamento do LPTMR

O LPTMR possui quatro registradores:

- **Control and Status Register (CSR)**: Nos permite configurar (*Timer Interrupt Enable*, TIE, bit 6) e verificar (*Timer Compare Flag*, TCF, bit 7) por interrupções, bem como iniciar e parar o contador (*Timer Enable*, TEN, bit 0).
- **Prescale Register (PSR)**: Nos permite selecionar um valor para o *prescaler* (PRESCALE, bits 3 a 6), ignorar o *prescaler* (PBYP, bit 2), e selecionar a fonte de *clock* (PCS, bits 0 e 1).
- **Compare Register (CMR)**: Registrador de 16-bits, que mantém o valor do módulo de contagem, utilizado para comparação com a contagem atual.
- **Counter Register (CTR)**: Registrador de 16-bits, que mantém a contagem.

Como mostra a figura 06 a seleção da fonte de *clock* para o LPTMR é feita utilizando os registradores **SOPT1** do SIM, configurando qual a fonte de *clock* do oscilador para o LPTMR

nos bits 18 e 19, e PSR do LPTMR, configurando a fonte de *clock* do LPTMR, nos bits 0 e 1. Caso seja utilizado o *clock* do oscilador, com valor 01 em PCS, devemos configurar o oscilador em SOPT1.

Podemos definir a sequência de inicialização do LPTMR (para a função de contagem de tempo) como se segue:

1. LPTMR deve ser desabilitado, se já estiver habilitado, com zero no bit 0 (TEN) de **CSR**.
2. Deve-se configurar o *prescaler*, nos bits 3 a 6 do registrador **PSR**, ou ativar o **bypass** (desvio), no bit 2 (PBYP).
3. Definir o módulo de comparação **CMR** usando a fórmula: MOD $= T_{overflow} \times \frac{f_{source}}{ps}$.
4. Habilitar as interrupções por *overflow* no bit 6 (TIE) no registrador **CSR**;
5. Habilite o Timer para iniciar a operação;

Os passos de habilitar o *clock* do periférico, habilitar a interrupção global no **NVIC** e configurar a **ISR** são omitidos em todos os temporizadores, já que são passos comuns a todos.

3.2 Atividade Prática com LPTMR usando Registradores

A função principal da prática, como mostra o código abaixo, assim como de todas as práticas, tem uma função para inicializar os LEDs e outra, ou outras, dependendo do número de módulos em uso, para inicializar os módulos do temporizador. O laço infinito prende a CPU, que só executará um processamento nas ISR.

```
1 int main(void) {
2   LEDS_Init();
3   LPTMR0_Init();
4   while(1);
5 }
```

Na função **LEDS_Init** fazemos a inicialização dos pinos utilizados para saída digital, para atuarem como LEDs. Desta forma, na linha 2 é ativado o *clock* dos controladores **PORT** dos pinos usados pelos LEDs. Nas linhas 4 e 5 configuramos o MUX para a função de GPIO. E nas linhas 7 e 8 definimos a direção do pinos para saída. Essa função será a mesma para todos os códigos que não usem SDK, portanto não será repetida.

```
1 void LEDS_Init(void) {
2   SIM->SCGC5 |= (1U << 10) | (1U << 12);
3
4   LED_01_PORT->PCR[3] |= (1U << 8);
```

```
5   LED_02_PORT->PCR[5] |= (1U << 8);
6
7   LED_01_GPIO->PDDR |= (1U << 3);
8   LED_02_GPIO->PDDR |= (1U << 5);
9 }
```

A função **LPTMR0_Init** irá configurar e iniciar o temporizador. Desta forma, temos na linha 1 a habilitação do *clock* para o periférico. Na linha 6 configuramos o periférico para não utilizar o *prescaler* e selecionamos o clock de 1 kHz proveniente do oscilador de baixa potência, o **LPO**. Na linha 10 definimos o valor de **TOP**, com o cálculo MOD = $1s \times 1kHz = 1000$, e sendo TOP = MOD - 1, e CMR = TOP, CMR = 999.

```
1  void LPTMR0_Init(void) {
2    // ENABLE CLOCK FOR LPTMR0
3    SIM->SCGC5 |= (1U << 0);
4
5    // BYPASS PRESCALER AND SELECT 1KHz LPO
6    LPTMR0->PSR |= (0b01U << 0) | (1U << 2);
7
8    // T = (1/1KHz) = 1ms
9    // CMR = 1s/1ms = 1000 COUNTS
10   LPTMR0->CMR = 999;
11
12   // ENABLE INTERRUPTS FOR LPTMR0 AND ENABLE IT
13   LPTMR0->CSR |= (1U << 0) | (1U << 6);
14
15   // ENABLE INTERRUPTS FOR LPTMR0 IN NVIC
16   NVIC->ISER[0] |= (1U << 28);
17 }
```

A **ISR** do periférico, que irá tratar a interrupção gerada por *overflow*, irá limpar a *flag* de pendente da interrupção na linha 3, e irá piscar os dois LEDs nas linhas 6 e 7.

```
1  void LPTMR0_IRQHandler(void) {
2    // CLEAR INTERRUPT FLAG
3    LPTMR0->CSR |= (1U << 7);
4
5    // BLINK THE TWO LEDS
6    LED_01_GPIO->PTOR |= (1U << 3);
7    LED_02_GPIO->PTOR |= (1U << 5);
8  }
```

Desta maneira, configuramos o LPTMR0 para gerar uma ISR a cada 1 segundo, que é o que é pedido no item A da pratica, tendo 1000 contagens, cada uma a 1ms.

Para o item B proposto devemos fazer o LED piscar à cada 8 segundos usando o prescaler. Desta maneira, devemos alterar apenas duas linhas:

- Na linha 6 desativamos o bit 2, o **PBYP** e selecionamos um valor de *prescaler*, nos bits 3 a 6, onde 0000 vale 2, 0001 vale 4, e etc, onde o valor do prescaler é $2^{PRESCALE+1}$.
- Na linha 10 devemos escolher um valor para o CMR que leve em conta o *prescaler*.

No nosso caso podemos realizar várias configurações diferente e ainda atingir o mesmo resultado. Iremos configurar o *prescaler* para dividir a frequência por 8, com o campo de bits **PRESCALE** igual a 0010.

Dessa forma, refazendo os cálculos, temos: MOD $= 8s \times \frac{1kHz}{8} = 1000$. Dessa forma, apenas modificando o valor do prescaler não precisamos modificar o valor de CMR (mantém-se em 999)!

Já no item C é pedido para piscar um LED a cada 2 segundos e outro a cada 6 segundos. Como só temos um módulo devemos utilizar a mesma contagem de forma que possamos controlar os dois LEDs. Tendo em vista que um LED irá piscar 2 vezes antes do outro LED piscar (os dois LEDs piscarão simultaneamente nesse momento), podemos estabelecer um contador, que conte quantas vezes se passaram 2 segundos. Quando esse contador atingir o valor 2 (iniciando de zero) teremos certeza que terão se passado 6 segundos e poderemos ligar o segundo LED. O contador, então, irá para zero. O primeiro LED piscará a cada chamada da ISR.

Para realizarmos a contagem de 2 segundos basta que modifiquemos o valor do *prescaler* para 2, não precisando modificar CMR, como mostra o cálculo: MOD $= 2s \times \frac{1kHz}{2} = 1000$.

A ISR em si é modificada com a adição da verificação do contador, a alternância do valor do LED é movida para essa estrutura de verificação e com a adição da soma circular o contador, na linha 11.

```
1  void LPTMR0_IRQHandler(void) {
2    // BLINK THE SECOND LED IN 6s
3    if(blinkCount == 2) {
4      LED_02_GPIO->PTOR |= (1U << 5);
5    }
6
7    // BLINK THE FIRST LED IN 2s
8    LED_01_GPIO->PTOR |= (1U << 3);
9
10   // GET THE CURRENT COUNTER
11   blinkCount = (blinkCount + 1) % 3;
12
13   // CLEAR INTERRUPT FLAG
14   LPTMR0->CSR |= (1U << 7);
15 }
```

3.3 Atividade Prática com LPTMR usando SDK

No último item da prática é proposto para implementarmos os itens A e B usando a API SDK. Com a mesma *main*, e alterando as funções, temos, para inicializar os pinos usados para os LEDs como saída:

```
1  void LEDS_Init(void) {
2    // ENABLE THE CLOCK FOR THE TWO LEDS PORTS
3    CLOCK_EnableClock(kCLOCK_PortB);
4    CLOCK_EnableClock(kCLOCK_PortD);
5
6    // SELECT GPIO FUNCTION FOR THE LEDS PINS
7    PORT_SetPinMux(PORTB, 3, kPORT_MuxAsGpio);
8    PORT_SetPinMux(PORTD, 5, kPORT_MuxAsGpio);
9
10   // CONFIG THE LED PINS AS OUTPUT
11   static const gpio_pin_config_t led01_handle = {kGPIO_DigitalOutput, 0};
12   GPIO_PinInit(GPIOB, 3, &led01_handle);
13   static const gpio_pin_config_t led02_handle = {kGPIO_DigitalOutput, 0};
14   GPIO_PinInit(GPIOD, 5, &led02_handle);
15 }
```

Abaixo está o código para a inicialização do temporizador. Após habilitarmos o *clock* do periférico devemos realizar sua configuração. Para isso utilizamos uma *struct*, do tipo **lptmr_config_t**, que mantém as definições do temporizador. Devemos inicializar uma variável deste tipo estruturado e repassar seu endereço para uma função que irá preenchê-la com as definições padrões para o periférico. Fazemos isso na linha 7, por meio da função **LPTMR_GetDefaultConfig**. Após isso podemos utilizar a variável para inicializar o periférico, repassando seu endereço para a função de inicialização, na linha 14, com a função **LPTMR_Init**, onde também informamos o endereço base do módulo utilizado.

Nesse caso o temporizador será configurado conforme as definições padrões. Caso seja necessária alguma configuração adicional ela deve ser feita antes da inicialização do periférico, modificando os campos da estrutura. No caso abaixo, obtemos a configuração padrão, e a modificamos na linha 10, para desabilitar o *prescaler*, e na linha 11, para selecionar a fonte de *clock* como o **LPO**. Esse é o procedimento padrão para todos os temporizadores (utilizando o SDK).

Com a função **LPTMR_SetTimerPeriod** configuramos o valor TOP, passando como argumento o endereço inicial do módulo e o valor de TOP, no caso 999, para 1000 contagens, totalizando 1 segundo.

```
1  void LPTMR0_Init(void) {
2    // ENABLE CLOCK FOR LPTMR0
3    CLOCK_EnableClock(kCLOCK_Lptmr0);
4
5    // GET THE DEFAULT CONFIG FOR THE LPTMR
6    static lptmr_config_t lptmr0_handle;
7    LPTMR_GetDefaultConfig(&lptmr0_handle);
8
9    // DIVIDE-BY-8 PRESCALER AND SELECT 1KHz LPO
```

```
10    lptmr0_handle.bypassPrescaler = true;
11    lptmr0_handle.prescalerClockSource = kLPTMR_PrescalerClock_1;
12
13    // INITIALIZES THE LPTMR PERIPHERAL
14    LPTMR_Init(LPTMR0, &lptmr0_handle);
15
16    // CONFIGURES THE NUMBER OF COUNTS UTIL OVERFLOW
17    // T = (1/1KHz) = 1ms
18    // CMR = 1s/1ms = 1000 TICKS
19    LPTMR_SetTimerPeriod(LPTMR0, 999);
20
21    // ENABLE INTERRUPTS FOR LPTMR0
22    LPTMR_EnableInterrupts(LPTMR0, kLPTMR_TimerInterruptEnable);
23
24    // START THE COUNT
25    LPTMR_StartTimer(LPTMR0);
26
27    // ENABLE INTERRUPTS FOR LPTMR0 IN NVIC
28    NVIC_EnableIRQ(LPTMR0_IRQn);
29  }
```

Abaixo é programada a ISR. Nela realizamos exatamente a mesma coisa que antes: limpamos a *flag* de pendente do módulo usando a função **LPTMR_ClearStatusFlags**, passando o endereço base do módulo e a constante **kLPTMR_TimerCompareFlag** para indicar que a *flag* a ser limpa é a de comparação do temporizador. Ao final o nível dos LEDs é alternado.

```
1  void LPTMR0_IRQHandler(void) {
2    // CLEAR INTERRUPT FLAG
3    LPTMR_ClearStatusFlags(LPTMR0, kLPTMR_TimerCompareFlag);
4
5    // TOGGLE THE LEDS
6    GPIO_TogglePinsOutput(GPIOB, (1U << 3));
7    GPIO_TogglePinsOutput(GPIOD, (1U << 5));
8  }
```

Para que possamos realizar um tempo de oito segundos devemos modificar apenas três linhas na função **LPTMR0_Init**. A linha 10 será omitida, para que o prescaler possa ser utilizado (valor padrão para o *bypass* é *false*, ou seja, prescaler ativado). Deveremos trocar essa linha pela seguinte linha de código:

```
1    lptmr0_handle.value = kLPTMR_Prescale_Glitch_2;
```

Dessa forma, escolhemos um *prescaler* de valor 8. Dessa forma, mantivemos 1000 contagens, não precisando modificar mais linhas, como já foi debatido.

Vale lembrar que a configuração de todos os periféricos temporizadores é semelhante a do LPTMR, e que as funções podem ser encontradas na biblioteca “fsl_lptmr.h”. Para outros periféricos é só substituir o nome do temporizador.

3.4 Vendo o Sinal Gerado com Osciloscópio

Na prática é pedido, no item C, que mostremos o sinal gerado, utilizando um osciloscópio, equipamento para ver sinais elétricos de corrente e tensão, utilizando-se de uma ponteira, como podemos ver na figura 07.

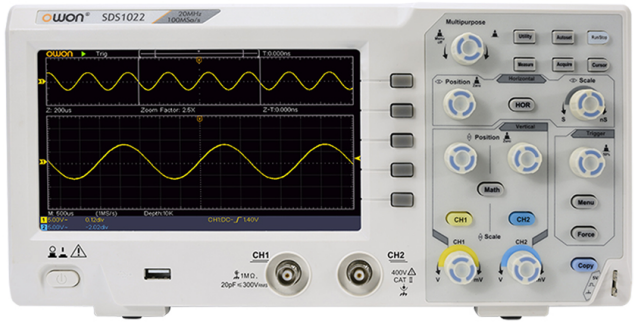

Figura 7: Osciloscópio, usado para verificar sinais elétricos

Podemos ver na figura 08 o sinal gerado em decorrência da configuração do temporizador LPTMR para 2 segundos. A cada *overflow* de contagem o sinal muda de nível, como especificado na ISR do LPTMR. O sinal de cima está com um período de 12 segundos, já que só muda de nível depois de 3 *overflows* de contagem.

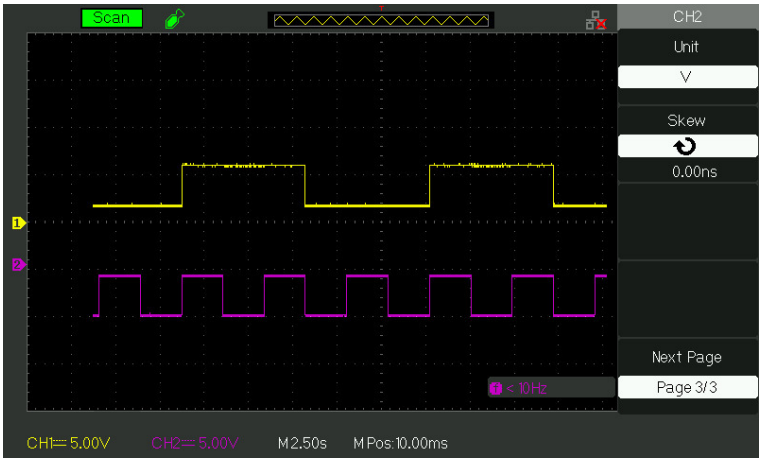

Figura 8: Sinal emitido usando o temporizador LPTMR

4 Periodic Interrupt Timer (PIT)

O periférico PIT, como mostra a figura 07, possui dois canais (0 e 1) e conta com um registrador contador de 32-bits. As características que mais se destacam neste módulo é sua capacidade de gerar disparos para o DMA, além de realizar contagens com interrupção por *overflow* e ter independência para cada canal.

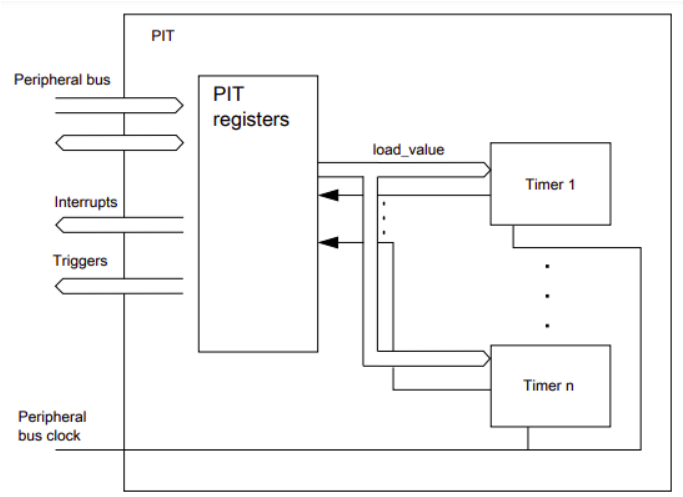

Figura 9: Diagrama de Blocos do PIT

4.1 Funcionamento do PIT

O funcionamento do nosso módulo acontece seguindo os seguintes passos: o primeiro item a ser configurado para inicializar o PIT é habilitar o Clock para o periférico. Essa operação é feita utilizando o registrador **System Clock Gating Control Register 6**, ou SCGC6, do módulo SIM, e ativando o bit 23.

Agora devemos configurar os parâmetros do PIT. Iremos, para isso, utilizar o registrador **PIT Module Control Register**, que tem como função ativar o temporizador e modo de depuração. Assim, é através do Bit **MDIS (Module Disable)** que é ativado, ou desativado, o temporizador.

O próximo passo é configurar o contador. Como já dito anteriormente, o PIT possui registradores de contagem de 32 bits, diferentemente do LPTMR e TPM. O registrador **Timer Load Value Register (PIT_LDVALn)** é responsável por receber o valor do módulo de contagem.

Logo após, devemos configurar o registrador **Timer Control Register PIT CTRLn** possui dois Bits a serem configurados, são eles: os Bit **TIE** (Timer Interrupt Enable) e Bit **TEN** (Timer Enable). E o último passo para ser realizar na inicialização do PIT é habilitar a interrupção, ativando o bit **TIE**, e habilitar o temporizador, ativando o bit **TEN**.

4.2 Atividade Prática com o PIT Utilizando os Registradores

Neste tópico vamos abordar o desenvolvimento das praticas proposta utilizando o temporizador PIT. Desta maneira, temos na função principal a chamada da função já comentada, **LEDS_Init**. A execução periódica das instruções da ISR acontece devido ao *overflow* periódico dos temporizadores. O código da prática está descrito abaixo:

```
1  int main(void) {
2    LEDS_Init();
3    PIT0_Init();
4    while(1);
5  }
```

Na função **PIT0_Init** vamos fazer a inicialização e configuração do temporizador. Desta forma, temos na linha 3 à habilitação do *clock* para o periférico. Em seguida na linha 6 é feito a habilitação do temporizador PIT. Na linha 9 é feito a seleção do canal zero e utiliza o registrador **LDVAL** para armazenar á contagem de 1 segundos. Na linha 12 é feito a configuração do registrador **TCTRL**, que faz a habilitação da *flag* de da interrupção. Na linha 15 temos a configuração do **ISER** que faz a habilitação de interrupções no NVIC para o PIT. Na linha 18 temos o **TCTRL** que faz a configuração de inicio de contagem.

```
1  void PIT0_Init(void) {
2    // ENABLE CLOCK FOR PIT
3    SIM->SCGC6 |= (1U << 23);
4
5    // ENABLE PIT TIMERS
6    PIT->MCR &= ~(1U << 1);
7
8    // SELECT 1s
9    PIT->CHANNEL[0].LDVAL = ((1*10485760) - 1);
10
11   // ENABLE INTERRUPTS FOR PIT
12   PIT->CHANNEL[0].TCTRL |= (1U << 1);
13
14   // ENABLE INTERRUPTS FOR PIT IN NVIC
15   NVIC->ISER[0] |= (1U << 22);
16
17   // START COUNT
18   PIT->CHANNEL[0].TCTRL |= (1U << 0);
19 }
```

Para configurarmos o tempo devemos colocar em LDVAL o número de contagens para realizar o tempo que queremos. O cálculo para esse válor está como se segue:

$$\text{LDVAL} = \text{Tempo em Segundos} \times 10485760 - 1$$

Abaixo está a ISR do PIT. A cada 1 segundo haverá uma interrupção por *overflow* de contagem, chamando a ISR e realizando a troca do nível dos LEDs.

```
1 void PIT_IRQHandler(void) {
2   if(PIT->CHANNEL[0].TFLG & (1U << 0)) {
3     // CLEAR INTERRUPT FLAG
4     PIT->CHANNEL[0].TFLG |= (1U << 0);
5
6     // BLINK THE TWO LEDS
7     LED_01_GPIO->PTOR |= (1U << 3);
8   }
9 }
```

Esta, então, foi a resolução do item A. Para a resolução do item B proposto precisamos apenas alterar a linha 9 da função de inicialização, colocando um valor de 83.886.079 para LDVAL. Com isso vamos gerar uma interrupção a cada 8 segundos que vai fazer o pino do LED trocar de nível.

Para resolução do item C vamos que ter configurar um segundo canal, como podemos ver no código abaixo. Para isso vamos simplesmente copiar a função **PIT0_Init** e realizar uma alteração na linha 9, para uma contagem de 6 segundos, e modificar os canais de 0 para 1. Teremos então duas funções de inicialização. A função do PIT0 deverá ser modificada para realizar o *overflow* a cada 2 segundos e devemos adicionar uma outra função, **PIT1_Init**, para o outro módulo. Dessa forma poderemos gerar um interrupção à cada 2 segundos e outra interrupção à cada 6 segundos.

```
1 void PIT1_Init(void) {
2   // ENABLE CLOCK FOR PIT
3   SIM->SCGC6 |= (1U << 23);
4
5   // ENABLE PIT TIMERS
6   PIT->MCR &= ~(1U << 1);
7
8   // SELECT 6s
9   PIT->CHANNEL[1].LDVAL =((6*10485760) - 1);
10
11  // ENABLE INTERRUPTS FOR PIT
12  PIT->CHANNEL[1].TCTRL |= (1U << 1);
13
14  // ENABLE INTERRUPTS FOR LPTMR0 IN NVIC
15  NVIC->ISER[0] |= (1U << 22);
16
17  // START COUNT
18  PIT->CHANNEL[1].TCTRL |= (1U << 0);
19 }
```

Devemos chamar, então, as duas funções de inicialização na main. Abaixo está a ISR do PIT. Devemos verificar qual canal do PIT que gerou a interrupção para podermos realizar o tratamento, que seria piscar um dos LEDs.

```
1 void PIT_IRQHandler(void) {
2   if(PIT->CHANNEL[0].TFLG & (1U << 0)) {
3     // CLEAR INTERRUPT FLAG
```

```
4       PIT->CHANNEL[0].TFLG |= (1U << 0);
5
6       // BLINK THE LED 1
7       LED_01_GPIO->PTOR |=(1U<<3);
8     }
9     else if(PIT->CHANNEL[1].TFLG & (1U << 0)) {
10      // CLEAR INTERRUPT FLAG
11      PIT->CHANNEL[1].TFLG |= (1U << 0);
12
13      // BLINK THE LED 2
14      LED_02_GPIO->PTOR |= (1U << 5);
15    }
16  }
```

4.3 Atividade Prática com o PIT Utilizando o SDK

No último item da prática é proposto para implementarmos os itens A e B usando a API SDK. Com a mesma *main*, onde chamamos as duas funções de inicialização do PIT.

Vamos realizar a inicialização do temporizador. Após habilitarmos o *clock* do periférico devemos realizar sua configuração. Para isso utilizamos uma *struct*, do tipo **pit_config_t**, sendo elas **pit0_handle** e **pit1_handle**, que mantém as configurações do temporizador. Devemos inicializar uma variável deste tipo estruturado e repassar seu endereço para uma função que irá preenchê-la com as definições padrões para o periférico. Fazemos isso na linha 6 e 28 por meio da função **PIT_GetDefaultConfig**. Após isso podemos utilizar a variável para inicializar o periférico, repassando seu endereço para a função de inicialização, na linha 8 e 30 com a função **PIT_Init**, onde também informamos o endereço base do módulo utilizado.

Nesse caso o temporizador será configurado conforme as definições padrões. Caso seja necessária alguma configuração adicional ela deve ser feita antes da inicialização do periférico, modificando os campos da estrutura. No caso abaixo obtemos a configuração padrão e a modificamos na linha 7 e 29, para habilitar o **PIT** mesmo em modo de depuração. Esse é o procedimento padrão para todos os temporizadores (utilizando o SDK).

```
1  void PIT0_Init(void) {
2    // ENABLE CLOCK FOR PIT
3    CLOCK_EnableClock(kCLOCK_Pit0);
4
5    static pit_config_t pit0_handle = {};
6    PIT_GetDefaultConfig(&pit0_handle);
7    pit0_handle.enableRunInDebug = true;
8    PIT_Init(PIT, &pit0_handle);
9
10   // SELECT 2s
11   PIT_SetTimerPeriod(PIT, 0, ((2*10485760) - 1));
12
13   // ENABLE INTERRUPTS FOR PIT
```

```
  PIT_EnableInterrupts(PIT, 0, kPIT_TimerInterruptEnable);

  // ENABLE INTERRUPTS FOR PIT IN NVIC
  NVIC_EnableIRQ(PIT_IRQn);

  // START COUNT
  PIT_StartTimer(PIT, 0);
}

void PIT1_Init(void) {
  // ENABLE CLOCK FOR PIT
  CLOCK_EnableClock(kCLOCK_Pit0);

  static pit_config_t pit1_handle = {};
  PIT_GetDefaultConfig(&pit1_handle);
  pit1_handle.enableRunInDebug = true;
  PIT_Init(PIT, &pit1_handle);

  // SELECT 8s
  PIT_SetTimerPeriod(PIT, 1, ((8*10485760) - 1));

  // ENABLE INTERRUPTS FOR PIT
  PIT_EnableInterrupts(PIT, 1, kPIT_TimerInterruptEnable);

  // ENABLE INTERRUPTS FOR LPTMR0 IN NVIC
  NVIC_EnableIRQ(PIT_IRQn);

  // START COUNT
  PIT_StartTimer(PIT, 1);
}
```

Abaixo é programada a ISR. Nela realizamos exatamente a mesma coisa que antes: limpamos a *flag* de pendente do módulo usando a função **PIT_ClearStatusFlags**, passando o endereço base do módulo e a constante **kPIT_TimerFlag** para indicar a *flag* a ser limpa. Ao final o nível dos LEDs é alternado.

```
void PIT_IRQHandler(void) {

  if(PIT_GetStatusFlags(PIT, 0) & kPIT_TimerFlag) {
    // CLEAR INTERRUPT FLAG
    PIT_ClearStatusFlags(PIT, 0, kPIT_TimerFlag);

    GPIO_TogglePinsOutput(GPIOB, (1U << 3));
  }
  else {
    // CLEAR INTERRUPT FLAG
    PIT_ClearStatusFlags(PIT, 1, kPIT_TimerFlag);

    GPIO_TogglePinsOutput(GPIOD,(1U << 5));
  }
}
```

Para que possamos realizar um tempo de um segundo, oito segundos ou outro tempo qualquer devemos apenas modificar o valor passado para a função **PIT_SetTimerPeriod**.

4.4 Vendo o Sinal Gerado com Osciloscópio

Podemos ver na figura 10 o sinal gerado em decorrência da configuração do temporizador PIT para 6 segundos e 2 segundos, respectivamente. A cada *overflow* de contagem o sinal muda de nível, como especificado na ISR do PIT.

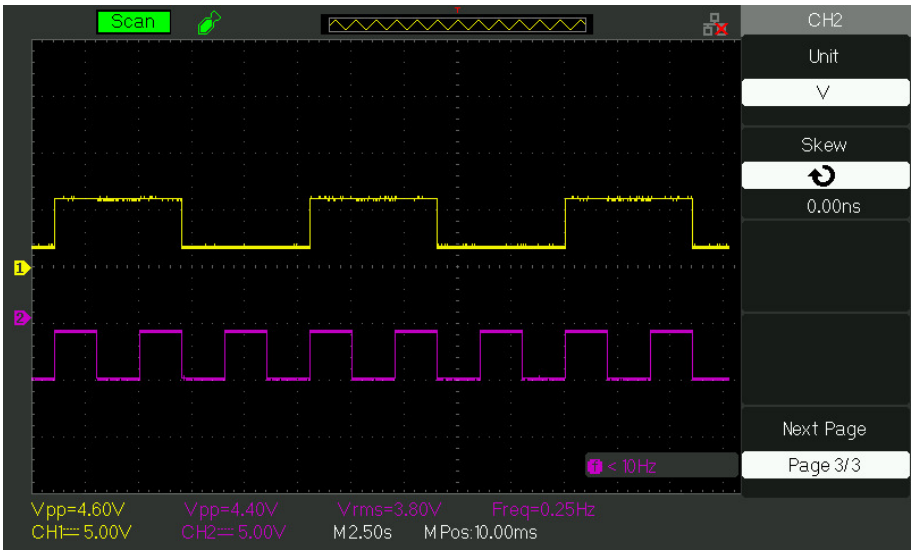

Figura 10: Sinal emitido usando o temporizador PIT

5 Timer/PWM Module (TPM)

O TPM (Módulo Timer/PWM) é um temporizador de 6 canais que suporta entrada, captura, comparação de saída e geração de sinais PWM, usados geralmente para controlar motores elétricos e gerenciar de o fornecimento de potência. O contador, os registradores de comparação e captura são cronometrados por um temporizador assíncrono que pode permanecer habilitado em modos de baixa potência. Desta forma, o mesmo possui os seguintes recursos:

- A *fonte de clock* do TPM é selecionável.
- Pode incrementar em cada borda do *clock* do contador assíncrono.
- A contagem pode incrementar na borda de subida de uma entrada de *clock* externa sincronizada com o contador assíncrono.
- *Prescaler* dividido por 1, 2, 4, 8, 16, 32, 64 ou 128.

• O TPM possui três módulos, cada um com 6 canais.

• TPM inclui um contador de 16 bits.

5.1 Funcionamento do TPM

A primeira configuração para inicializar o periférico TPM é o System Clock Gating Control Register 6 (SCGC6). Como foi comentado anteriormente, o microcontrolador da placa KL25Z possui três módulos (TPM0, TPM1 e TPM2). Desta forma, para ativar cada módulo, um bit específico é escrito no SCGC6, configurando os bits 24, 25 e 26 para TPM0, TPM1 e TPM2, respectivamente.

Outro item importante a ser configurado para utilizar o TPM é sua fonte de *clock*. Para esta tarefa, o registrador do **System Option Register 2** (SOPT2) deve ser configurado nos bits 24 e 25. As opções de fontes de *clock* para o TPM são mostradas na figura 11.

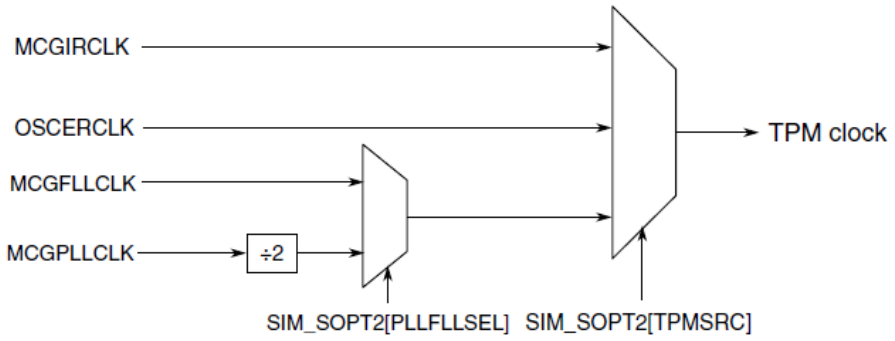

Figura 11: Fontes de *clock* disponíveis para o TPM

Vale ressaltar que o TPM possui três fontes de clock que são: **MCGFLLCLK or MCGPLLCLK/2** (*clocks* do barramento ou do núcleo dividido por 2) com 01, **OSCERCLK** (*clock* do oscilador externo) com 10 e **MCGIRCLK** (*clock* de referência externo) com 11. Se quisermos selecionar o *clock* **MCGFLLCLK** ou **MCGPLLCLK/2** devemos configurar o bit 16 do registrador SOPT2, sendo 0 e 1 respectivamente.

Queremos utilizar a frequência de 32,768 kHz, provinda da fonte de *clock* MCGIRCLK. Para isso devemos selecionar no SOPT2, com 11 nos bits 24 e 25, e configurar 2 registradores do módulo **MCG** (Multipurpose Clock Generator). Devemos ativar o bit 1 (**IRCLKEN**) do registrador **C1** para ativar o *clock* de referência interno. Devemos agora selecionar o *clock* lento, desativando o bit 0 (IRCS) do registrador C2, já que o MCGIRCLK também pode emitir um sinal de *clock* com uma frequência de 4 MHz (*clock* rápido). Dessa forma podemos usar um clock de 32,768 kHz no TPM.

Devemos agora configurar o módulo de contagem no registrador **MOD**, que contém o valor do módulo para o contador TPM, que seria TOP - 1. Quando o contador TPM atinge o valor do módulo e incrementa, o sinalizador de *overflow* (TOF) é emitido e o próximo valor do

contador TPM será 0 antes de reiniciar a contagem. Devemos também habilitar a interrupção no bit 6 (**TOIE**) do registrador SC e configurar o *prescale*, nos primeiros 3 bits do registrador SC, sendo 000 divisão por 1 e 111 divisão por 128. Ao final devemos habilitar o temporizador, com 01 nos bits 3 e 4 (**CMOD**), para habilitar a contagem a cada pulso de *clock*.

5.2 Atividade Prática com o TPM Utilizando os Registradores

A função principal da prática, como mostra o código abaixo, assim como de todas as práticas, tem uma função para inicializar os LEDs e abaixo dela temos a configuração dos módulos do TPM e da ISR que irá tratar a IRQ lançada em *overflow*. Para a prática A e B iremos utilizar apenas um módulo, e para a prática C iremos utilizar os dois. Devemos, então, chamar a função **TPM1_Init**.

```
1 int main(void) {
2   LEDS_Init();
3   TPM0_Init();
4   while(1);
5 }
```

A função **TPM0_Init** irá configurar e iniciar o temporizador. Desta forma, temos nas linha 3 e 6 a habilitação do *clock* **MCGIRCLK** em C1 e a habilitação da frequência de 32 kHz em C2. Na linha 9 configuramos o registrador **SOPT2** para habilitar um clock de 32,768 KHz. Na linha 14 vamos habilitar a *flag* de interrupção, deixando o *prescale* como 1 (valor padrão). Seguindo temos na linha 17 a configuração do modulo da contagem para 1 segundos, como 32768. E por fim temos na linha 20 habilitação do **ISER**, que faz habilitação de interrupções no NVIC para o TPM e habilitamos a contagem no registrador SC.

```
1  void TPM0_Init(void) {
2    // ENABLE MCGIRCLK.
3    MCG->C1 |= (1U << 1);
4
5    //SELECT 32,768KHz FREQUENCY FOR MCGIRCLK
6    MCG->C2 &= ~(1U << 0);
7
8    // SELECT 32,768KHz.
9    SIM->SOPT2 |= (0b11U << 24);
10
11   // ENABLE CLOCK FOR TPM0
12   SIM->SCGC6 |= (1U << 24);
13
14   TPM0->SC |= (1U << 6);
15
16   // MOD = 1s * (32,768KHz/1) = 32768
17   TPM0->MOD = 32767;
18
19   // ENABLE INTERRUPTS FOR TMP0 IN NVIC
20   NVIC->ISER[0] |= (1U << 17);
```

```
21
22   // ENABLE TPM TO COUNT
23   TPM0->SC |= (0b01U << 3);
24 }
```

A **ISR** do periférico, que irá tratar a interrupção gerada por *overflow*, irá limpar a *flag* que está pendente na interrupção na linha 3, e irá piscar os dois LEDs na linha 6.

```
1 void TPM0_IRQHandler(void) {
2   // CLEAR INTERRUPT FLAG
3   TPM0->SC |= (1U << 7);
4
5   // BLINK THE TWO LEDS
6   LED_01_GPIO->PTOR |= (1U << 3);
7   LED_02_GPIO->PTOR |= (1U << 5);
8 }
```

Desta maneira, configuramos o TPM0 para gerar uma interrupção a cada 1 segundo, que é o que é pedido no item A da pratica, tendo 32768 contagens.

Para o item B proposto devemos fazer o LED piscar à cada 8 segundos usando *prescale*. Para isso devemos apenas modificar as linha 14 e 17 da função de inicialização do TPM0. Devemos utilizar um *prescale* de 32:

```
1 // Em TPM0_Init:
2 TPM0->SC |= (0b101U << 0);
```

E devemos configurar um módulo para 8 segundos. Para um segundo temos um módulo de 1024, já que:

$$\text{MOD} = 1\text{s} \times (32{,}768\ \text{kHz}/32) = 1024$$

Devemos então apenas multiplicar esse valor pelo tempo que queremos, no caso, 8:

```
1 // Em TPM0_Init: 8*1024 - 1
2 TPM0->MOD |= 8191;
```

Para a realização do item C iremos utilizar dois módulos do TPM. Dessa forma iremos configurar o TPM0 e TPM1(em uma função que será semelhante a TPM0, mudando o valor do módulo, também sendo chamada na *main*) para gerar uma interrupção a cada 2 e 6 segundos, respectivamente.

As duas funções de inicialização serão idênticas, diferindo apenas no valor do módulo:

```
1 // Em TPM0_Init:
2 TPM0->MOD |= 2047;
3
```

```
// Em TPM1_Init:
TPM1->MOD |= 6143;
```

Devemos também incluir os tratadores de interrupção de ambos os módulos:

```
void TPM0_IRQHandler(void) {
  // CLEAR INTERRUPT FLAG
  TPM0->SC |= (1U << 7);

  // BLINK THE LED
  LED_01_GPIO->PTOR |= (1U << 3);
}

void TPM1_IRQHandler(void) {
  // CLEAR INTERRUPT FLAG
  TPM1->SC |= (1U << 7);

  // BLINK THE LED
  LED_02_GPIO->PTOR |= (1U << 5);
}
```

5.3 Atividade Prática com o TPM Utilizando o SDK

No último item da prática é proposto para implementarmos os itens A e B usando a API SDK. Com a mesma *main*, e alterando as funções.

Iremos realizar a inicialização do temporizador. Após habilitarmos o *clock* do periférico devemos realizar sua configuração. Para isso utilizamos uma *struct*, do tipo **tpm_config_t**, que mantém as definições do temporizador. Devemos inicializar uma variável deste tipo estruturado e repassar seu endereço para uma função que irá preenchê-la com as definições padrões para o periférico. Fazemos isso na linha 11 e 40, por meio da função **TPM_GetDefaultConfig**. Após isso podemos utilizar a variável para inicializar o periférico, repassando seu endereço para a função de inicialização, na linha 13 e 42, com a função **TPM_Init**, onde também informamos o endereço base do módulo utilizado.

Nesse caso o temporizador será configurado conforme as definições padrões. Caso seja necessária alguma configuração adicional ela deve ser feita antes da inicialização do periférico, modificando os campos da estrutura. Esse é o procedimento padrão para todos os temporizadores (utilizando o SDK).

Na linha 5 é realizada a configuração do MCGIRCLK, habilitando-o e selecionando a frequência de 32,768 kHz, e na linha 16 é selecionada a opção 3, referente ao MCGIRCLK.

Com a função **TPM_SetTimerPeriod** configuramos o valor do módulo, passando como argumento o endereço inicial do módulo e o valor do módulo de contagem, no caso 1023, para 1024 contagens, totalizando 1 segundo. Para realizar a contagem de 8 segundos usaremos o outro módulo do TPM, o configurando na função **TPM1_Init**, apenas alterando o valor do

módulo para 8191.

```
void TPM0_Init(void) {
  // CONFIGURE THE INTERNAL REFERENCE CLOCK TO MCGIRCLK.
  // Slow internal reference clock selected
  // Fast IRC divider: divided by 1
  CLOCK_SetInternalRefClkConfig(kMCG_IrclkEnable, kMCG_IrcSlow, 0x0U);

  CLOCK_EnableClock(kCLOCK_Tpm0);

  // SET A DIVIDE-BY-32 PRESCALER
  static tpm_config_t tpm0_handle = {};
    TPM_GetDefaultConfig(&tpm0_handle);
    tpm0_handle.prescale = kTPM_Prescale_Divide_32;
    TPM_Init(TPM0, &tpm0_handle);

  // ENABLE CLOCK FOR TPM0
    CLOCK_SetTpmClock(3);

  // MOD = 2s * (32,768KHz/32) = 2048
    TPM_SetTimerPeriod(TPM0, 2047);

    TPM_EnableInterrupts(TPM0, kTPM_TimeOverflowInterruptEnable);

  // ENABLE INTERRUPTS FOR TMP0 IN NVIC
    NVIC_EnableIRQ(TPM0_IRQn);

  // ENABLE TPM TO COUNT
    TPM_StartTimer(TPM0, kTPM_SystemClock);
}

void TPM1_Init(void) {
  // CONFIGURE THE INTERNAL REFERENCE CLOCK TO MCGIRCLK.
  // Slow internal reference clock selected
  // Fast IRC divider: divided by 1
  CLOCK_SetInternalRefClkConfig(kMCG_IrclkEnable, kMCG_IrcSlow, 0x0U);

  CLOCK_EnableClock(kCLOCK_Tpm1);

  // SET A DIVIDE-BY-32 PRESCALER
  static tpm_config_t tpm1_handle = {};
    TPM_GetDefaultConfig(&tpm1_handle);
    tpm1_handle.prescale = kTPM_Prescale_Divide_32;
    TPM_Init(TPM1, &tpm1_handle);

  // ENABLE CLOCK FOR TPM0
    CLOCK_SetTpmClock(3);

  // MOD = 8s * (32,768KHz/32) = 8191
    TPM_SetTimerPeriod(TPM1, 8191);

    TPM_EnableInterrupts(TPM1, kTPM_TimeOverflowInterruptEnable);

```

```
52    // ENABLE INTERRUPTS FOR TMP0 IN NVIC
53      NVIC_EnableIRQ(TPM1_IRQn);
54
55    // ENABLE TPM TO COUNT
56      TPM_StartTimer(TPM1, kTPM_SystemClock);
57  }
```

Abaixo é programada as duas ISR. Nela realizamos exatamente a mesma coisa que antes: limpamos a *flag* de pendente do módulo usando a função **TPM_ClearStatusFlags**, passando o endereço base do módulo e a constante **GPIO_TogglePinsOutput** para indicar que a *flag* a ser limpa é a de comparação do temporizador. Ao final o nível dos LEDs é alternado.

```
1  void TPM0_IRQHandler(void) {
2    TPM_ClearStatusFlags(TPM0, kTPM_TimeOverflowFlag);
3    GPIO_TogglePinsOutput(GPIOB, (1U << 3));
4  }
5
6  void TPM1_IRQHandler(void) {
7    TPM_ClearStatusFlags(TPM1, kTPM_TimeOverflowFlag);
8    GPIO_TogglePinsOutput(GPIOD, (1U << 5));
9  }
```

5.4 Vendo o Sinal Gerado com Osciloscópio

Podemos ver na figura 10 o sinal gerado em decorrência da configuração do temporizador TPM para 6 segundos e 2 segundos, respectivamente. A cada *overflow* de contagem o sinal muda de nível, como especificado na ISR do TPM.

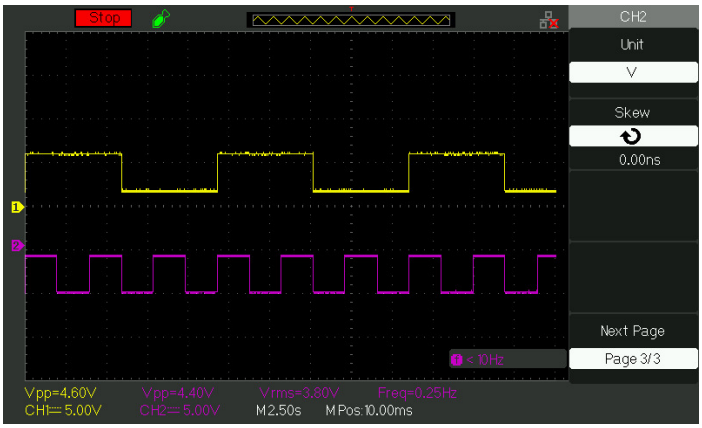

Figura 12: Sinal emitido usando o temporizador TPM

6 Conclusão

Neste trabalho discutimos e colocamos em prática os conceitos e métodos necessários para configurar os temporizadores da placa KL25Z, mas especificamente o: LPTMR, o PIT e o TPM. Com isso obtivemos um conhecimento mais avançado na configuração de temporizadores, podendo utilizar temporizadores de outro microcontroladores com mais facilidade, apenas adequando os conhecimentos adquiridos.

Pudemos também entender aplicações de temporizadores e como eles podem ser usados em um sistema, já que o conceito de **tempo** em um sistema embarcado é muito importante.

No mais, agradecemos ao professor pelos ensinamentos e desafios dessa atividade, que propôs melhoria em nosso conhecimento acadêmico.

Referências

[1] Repositório da disciplina "microcontroller_practices". https://github.com/pedrobotelho15/microcontroller_practices. Accessed: 2022-04-06.