

# 7

# Temporizadores/ Contadores (*Timers*)

Versão 1.0, Versão 2.0 (16/03/2020), Versão 3.0 (25/03/2020-Covid-19)

Nas aplicações com microcontroladores, é comum a necessidade de se marcar intervalos tempo, gerar sinais com certa periodicidade, medir a duração de eventos e controlar saídas com Modulação por Largura de Pulso (PWM – *Pulse Width Modulation*). É para atender à essas demandas que os microcontroladores trazem um recurso denominado Temporizador/Contador (TC - *Timer/Counter*), que é assunto deste capítulo. De agora em diante, para facilitar a leitura, as palavras contador e temporizador ou simplesmente timer serão usadas como sinônimos.

Para atender à necessidade de temporização, o MSP430 F5529, que é motivo deste estudo oferece os seguintes recursos:

- *Timer A0*, instância 0 do *Timer A*, com 5 comparadores;
- *Timer A1*, instância 1 do *Timer A*, com 3 comparadores;
- *Timer A2*, instância 2 do *Timer A*, com 3 comparadores e
- *Timer B0*, instância 0 do *Timer B*, com 7 comparadores.

Este capítulo não aborda o Relógio de Tempo Real (*Real-time Clock*) e nem o Temporizador Cão-de-Guarda (*Watchdog Timer*). Somente serão tratados os *Timers A* e *B* e suas instâncias.

## 7.0. Quero Usar os *Timers* e não Pretendo Ler Todo este Capítulo

O MSP430, na versão F5529 tem disponível as instâncias dos *timers A* e *B*, nomeadas TA0, TA1, TA2 e TB0. Como a instância do timer *B* é única, ela será sempre referenciada como TB0. Os dois timers (TA e TB) serão abordados como se fossem idênticos. As particularidades de TB serão apresentadas no final deste tópico. Cada instância tem um

contador de 16 bits: TA0R, TA1R, TA2R e TB0R e vários registradores de comparação e captura (CC), também de 16 bits, numerados como TA0CCR0, TA0CCR1, ..., TA1CCR0 etc. A tabela abaixo elenca todos os registradores dos *timers* disponíveis nesta versão do MSP. São usados os nomes genéricos **TAxCCRn**, e **TAxCCTLn**, onde:

- a letra “x” indica uma instância do *timer* ( $x = 0, 1, \dots$ ) e
- a letra “n” indica uma instância da unidade captura e comparação ( $n = 0, 1, \dots$ ).

Mais compacta ainda é a notação **TAx.n** ou **TB0.n**. Veja os exemplos:

- **TA0.1** → *timer* A0, instância 1 da unidade de captura e comparação;
- **TA2.2** → *timer* A2, instância 2 da unidade de captura e comparação;
- **TB0.3** → *timer* B0, instância 3 da unidade de captura e comparação;

*Tabela com todos os registradores dos timers*

<b>TA0</b>		<b>TA1</b>		<b>TA2</b>		<b>TB0</b>	
Operação	Controle	Operação	Operação	Operação	Controle	Operação	Controle
TA0R	TA0CTL	TA1R	TA1R	TA2R	TA2CTL	TB0R	TB0CTL
TA0CCR0	TA0CCTL0	TA1CCR0	TA1CCR0	TA2CCR0	TA2CCTL0	TB0CCR0	TB0CCTL0
TA0CCR1	TA0CCTL1	TA1CCR1	TA1CCR1	TA2CCR1	TA2CCTL1	TB0CCR1	TB0CCTL1
TA0CCR2	TA0CCTL2	TA1CCR2	TA1CCR2	TA2CCR2	TA2CCTL2	TB0CCR2	TB0CCTL2
TA0CCR3	TA0CCTL3	-	-	-	-	TB0CCR3	TB0CCTL3
TA0CCR4	TA0CCTL4	-	-	-	-	TB0CCR4	TB0CCTL4
-	-	-	-	-	-	TB0CCR5	TB0CCTL5
-	-	-	-	-	-	TB0CCR6	TB0CCTL6
-	TA0EX0	-	TA1EX0	-	TA2EX0	-	TB0EX0
-	TA0IV	-	TA1IV	-	TA2IV	-	TB0IV

*Resumo dos principais registradores dos timers*

<b>Finalidade</b>	<b>Registrador</b>	<b>Controle</b>	<b>Flag IFG</b>
Contador	TAxR (TB0R)	TAxCTL (TB0CTL)	TAIFG
Captura/Compara	TAxCCRn (TB0CCRn)	TAxCCTLn (TB0CCTLn)	CCIFG

A interação com as unidades de captura e comparação de cada *timer* é feita por meios de alguns pinos do processador. A tabela abaixo apresenta a lista completa.

*Tabela com a distribuição dos pinos de GPIO para os diferentes timers, para serem usados na captura de eventos ou na geração de saídas, na versão F5529  
(em cinza estão os pinos não disponíveis no Launch Pad)*

<b>Timer A0</b>		<b>Timer A1</b>		<b>Timer A2</b>		<b>Timer B0</b>	
TA0.0	P1.1	TA1.0	P1.7	TA2.0	P2.3	TB0.0	P5.6
TA0.1	P1.2	TA1.1	P2.0	TA2.1	P2.4	TB0.1	P5.7
TA0.2	P1.3	TA1.2	P2.1	TA2.2	P2.5	TB0.2	P7.4
TA0.3	P1.4	-	-	-	-	TB0.3	P7.5
TA0.4	P1.5	-	-	-	-	TB0.4	P7.6
-	-	-	-	-	-	TB0.5	P3.5
-	-	-	-	-	-	TB0.6	P3.6

O contador de cada *timer* (TAxR ou TB0R) conta de forma ascendente usando o relógio selecionado pelo usuário. É possível selecionar o ACLK (32.768 Hz) ou o SMCK (1.048.576 Hz) ou ainda um relógio externo apresentado em um pino específico do MSP, como listado na tabela abaixo. O usuário pode especificar uma série de divisores (até 1/64, vide Figura 7.16) para diminuir a frequência do relógio selecionado.

*Tabela com as entradas para relógio (*clock*) externo na versão F5529  
(Pino P1.0 é usado com o led vermelho e o pino P7.7 não está acessível na LaunchPad)*

<b>Clock externo</b>	<b>Pino</b>
TA0CLK	P1.0
TA1CLK	P1.6
TA2CLK	P2.2
TB0CLK	P7.7

Para cada contador (TAxR ou TB0R), o usuário pode selecionar 4 modos de contagem:

- **Modo 0**, Parado (*Stop*) → Serve para parar o contador.
- **Modo 1**, Ascendente (*Up*) → O contador TAxR (TB0R) conta de forma ascendente até seu valor ficar igual ao de TAxCCR0 (TB0CCR0), quando então recomeça a contagem a partir do zero.
- **Modo 2**, Contínuo (*Continuous*) → O contador TAxR (TB0R) conta de forma ascendente até seu valor máximo (0xFFFF) e recomeça a contagem a partir do zero. O contador TB0R tem seu tamanho configurável para 8, 10, 12 ou 16 bits e por isso seu máximo é 0xFF, 0x3F, 0xFFF ou 0xFFFF, respectivamente.

- **Modo 3, Sobe/Desce (Up/Down)** → O contador TAxR (TB0R) conta de forma ascendente até seu valor ficar igual ao de TAxCCR0 (TB0CCR0), quando então passa a contar de forma descendente até chegar a zero e recomeçar a contagem.

Gerar saídas (PWM): Cada unidade de captura e compara de cada *timer* pode selecionar um mecanismo para gerar saídas. Basicamente, são dois os momentos em que a saída é alterada, como listado abaixo. A próxima tabela indica as figuras que apresentam as possibilidades de contagem e geração de saída.

- TAxR (TB0R) vai a zero e
- TAxR (TB0R) coincide com um dos registradores de captura e comparação.

*Tabela indicando as figuras que deverão ser consultadas para cada modo de operação e modo de geração de saídas*

Modo	Contagem	Saída
Modo 0	Parado	Parado
Modo 1	7.11	7.19
Modo 2	7.14	7.20
Modo 3	7.15	7.21

Capturar evento: Um evento é caracterizado por um determinado pino de GPIO mudando de estado. Pode ser um flanco de subida, um flanco de descida ou ambos. Quando acontece o evento especificado, o valor do contador (TAxR ou TB0R) é copiado para uma das unidades de captura e compara. Isso fornece precisão para caracterizar o instante em que o evento aconteceu.

Para a operação dos *timers*, duas *flags* são muito importantes.

- TAIFG (TBIFG) → vai a “1” toda vez que o contador passar pelo zero e
- CCIFG → vai a “1” toda vez que o valor do contador coincidir com o valor do registrador de comparação, ou vai a “1” para indicar que aconteceu uma captura.

Essas duas *flags* podem provocar as interrupções listadas na tabela abaixo. Para que cada interrupção aconteça, é preciso realizar uma habilitação específica. Os vetores de interrupção estão listados na tabela a seguir.

- CCIFG, habilitar com CCIE = 1 no registrador correspondente e
- TAIFG (TBIFG), habilitar TAIE (TBIE) no registrador de controle correspondente.

*Tabela com os vetores de interrupção para os timers*

<b>Timer</b>	<b>Flag (IFG)</b>	<b>Vetor</b>	<b>Vetor (constante)</b>
TB0.0	TB0CCTL0.CCIFG	59	TIMER_B0_VECTOR
TB0.1, ... TB0.4	TB0CCTL1.CCIFG, ..., TB0CCTL6.CCIFG, TB0CTL.TAIFG	58	TIMER_B1_VECTOR
TA0.0	TA0CCTL0.CCIFG	53	TIMER0_A0_VECTOR
TA0.1, ... TA0.4	TA0CCTL1.CCIFG, ..., TA0CCTL4.CCIFG, TA0CTL.TAIFG	52	TIMER0_A1_VECTOR
TA1.0	TA1CCTL0.CCIFG	49	TIMER1_A0_VECTOR
TA1.1, TA1.2	TA1CCTL1.CCIFG, TA1CCTL2.CCIFG, TA1CTL.TAIFG	48	TIMER1_A1_VECTOR
TA2.0	TA2CCTL0.CCIFG	44	TIMER2_A0_VECTOR
TA2.1, TA2.2	TA2CCTL1.CCIFG, TA2CCTL2.CCIFG, TA2CTL.TAIFG	43	TIMER2_A1_VECTOR

Para finalizar, é preciso indicar que o *Timer B* apresenta as seguintes particularidades:

- O tamanho do contador (TB0R) é configurável para 8, 10, 12 ou 16 bits;
- Os registradores TB0CCRn têm dupla buferização e podem ser agrupados;
- Todas as saídas do *Timer B0* podem entrar no estado de alta-impedância e
- O bit SCCI não está disponível.

Vamos agora ver exemplos de possíveis empregos dos *timers*. Vamos sempre usar TA0 como exemplo.

### Quero uma interrupção periódica

Uma solução simples é empregar o *timer* no modo 1 (*Up*) e usar o comparador para limitar a contagem e, ao mesmo tempo, ativar uma flag de interrupção (TAIFG ou CCIFG). O exemplo é para uma interrupção a cada 1 ms usando o SMCLK (1.048.576 Hz). Em 1 ms temos 1.048,5 contagens. Aproximamos para 1.049, mas como o zero é contado, usaremos uma contagem a menos. Ficamos então com TA0CCR0 = 1.048.

```
//Configuração de TA0
TA0CTL = TASSEL_2 | MC_1;    //SMCLK e Modo 1
TA0CCTL0 = CCIE;           //CCIFG interrompe
TA0CCR0 = 1048;            //Limite de contagem (1 ms)
__enable_interrupt();       //Habilitação geral (GIE = 1)
...
// Rotina para atender à interrupção
#pragma vector = 53
__interrupt void isr_ta0_ifg(void){...}
```

### Preciso gerar um atraso

Usar o modo 1 (Up) e esperar a *flag* TAIFG ir para 1 (contador é zerado). Exemplo para gerar um atraso de 10 ms com SMCLK (1.048.576). Em 10 ms temos 10.485,76 contagens. Arredondando e levando em conta que o zero é também contado, usamos 10.485.

```
//Configuração de TA0
TA0CTL = TASSEL_2 | MC_1 | TAIE; //SMCLK, Modo 1 e TAIFG interrompe
TA0CCR0 = 10485;                //Limite de contagem (10 ms)
...
// Repetir cada vez que necessitar do atraso de 10 ms
TA0CTL |= TACLR;               //Zerar o contador
TA0CTL &= ~TAIFG;              //Apagar passado de TAIFG
while ( (TA0CTL&TAIFG) == 0); //Esperar 10 ms
```

### Preciso gerar um PWM

Um PWM é especificado pelo período e ciclo de carga. O ciclo de carga é variável e aqui vamos considerá-lo variando de 0% até o máximo 100%. Como exemplo, vamos especificar o período do PWM em 10 ms (100 Hz), o que corresponde a 10.485 contagens. É preciso escolher uma das saídas disponíveis (consultar tabela já vista). Usaremos o pino P1.2 que está designado para TA0.1. A saída fará uso do Modo 6 (Figura 7.19).

```
//Configuração de TA0 e P1.2
TA0CTL = TASSEL_2 | MC_1;    //SMCLK e Modo 1
TA0CCR0 = 10485;             //Limite de contagem (10 ms -> 100 Hz)
TA0CCTL1 = OUTMOD_6;         //Modo da saída
P1DIR |= BIT2;               //P1.2 como saída
P1SEL |= BIT2;               //Dedicado para TA0.1
...
// A variável int cc especifica o ciclo de carga em percentagem
// Cada vez que ela for alterada, é preciso atualizar valor de TA0CCR1
```

```
TA0CCR1 = 10485*(cc/100.); //Notar que a conta é feita em float (100.)
```

Para o caso de se acionar motores DC, a melhor solução é com o contador no Modo Sobe/Desce porque nas alterações do ciclo de carga, os pulsos ficam mais bem posicionados. Refazendo o exemplo acima.

```
//Configuração de TA0 e P1.2
TA0CTL = TASSEL_2 | MC_3; //SMCLK e Modo 1
TA0CCR0 = 10485/2; //5.242 subindo e 5.242 descendo (10ms)
TA0CCTL1 = OUTMOD_6; //Modo da saída
P1DIR |= BIT2; //P1.2 como saída
P1SEL |= BIT2; //Dedicado para TA0.1
...
// A variável int cc especifica o ciclo de carga em percentagem
// Cada vez que ela for alterada, atualizar valor de TA0CCR1
TA0CCR1 = 10485*(1-(cc/100.)); //Notar que a conta é feita em float
```

## Preciso capturar um evento

O evento a ser capturado precisa ser caracterizado pela alteração num dos pinos com esta destinação. Vamos exemplificar a captura medindo o período de uma onda quadrada que chega ao pino P1.2 (TA0.1) e escrevendo sua frequência na variável `float freq`. É claro que o relógio do contador deve ser coerente com a “velocidade” do evento que se quer capturar. O modo 2 é o mais indicado, pois assim o contador tem a maior excursão possível. Neste exemplo será usado o *polling* de CCIFG, mas também pode ser usada a interrupção.

```
// Variáveis que serão usadas no exemplo
unsigned int cap1, cap2; //Receber o valor da captura
long dif; //Receber diferença entre as capturas
float freq; //Valor da frequência
...
//Configuração de TA0 e P1.2
TA0CTL = TASSEL_2 | MC_2; //SMCLK, Modo 2
TA0CCTL1 = CM_1 | CCIS_0 | SCS | CAP; //Captura flanco de subida
P1DIR &= ~BIT2; //P1.2 como entrada
P1REN |= BIT2; //Habilitar resistor
P1OUT |= BIT2; //Para pullup
P1SEL |= BIT2; //Dedicado para captura (TA0.1)
...
// Esperar pela primeira captura
TA0CCTL1 &= ~CCIFG; //Apagar CCIFG
while ( (TA0CCTL1&CCIFG) == 0); //Esperar primeiro flanco
TA0CCTL1 &= ~CCIFG; //Apagar CCIFG
cap1 = TA0CCR1; //Ler primeira captura
;
```

```

while ( (TA0CCTL1&CCIFG) == 0); //Esperar segundo flanco
TA0CCTL1 &= ~CCIFG;           //Apagar CCIFG
cap2 = TA0CCR1;              //Ler segunda captura
;
// Calcular a frequência
dif = cap2 - cap1;           //Calcular a diferença entre as capturas
if (dif<0) dif += 65536L;     //Se negativa, ajustar
freq = 1048576./dif;         //Calcular a frequência, conta em float

```

Este resumo apresentou uma ideia muito simples dos timers. É recomendada o estudo dos exemplos apresentados no final deste capítulo. Em todo o caso, é considerada muito proveitosa a leitura de todo este capítulo.

## 7.1. Introdução aos *Timers*

Este tópico faz a construção passo a passo de um temporizador simplificado. A intenção é esclarecer e consolidar os conceitos mais importantes. Pode parecer estranho ao leitor os nomes usados neste tópico 7.1, porém a intenção foi a de buscar familiaridade com os nomes usados no MSP. Se o leitor já é familiarizado com o assunto, pode avançar para o próximo tópico.

Um temporizador (ou *timer*) nada mais é que um contador digital cuja frequência de contagem é controlada pelo programador. No caso do MSP430 este contador é de 16 *bits*, como exemplificado na Figura 7.1. Nesta figura temos um contador denominado de TxR, cuja frequência de contagem é especificada pelo sinal TxCLK. Como se pode ver na figura, a qualquer momento, é possível escrever ou ler neste contador TxR.

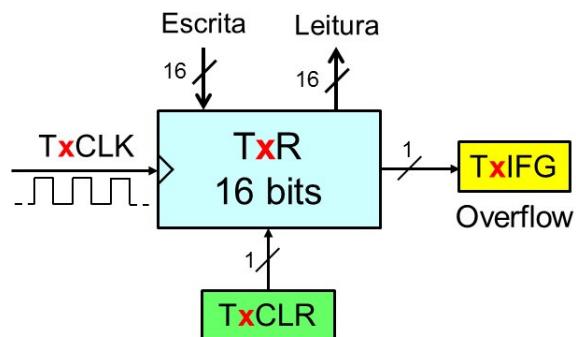
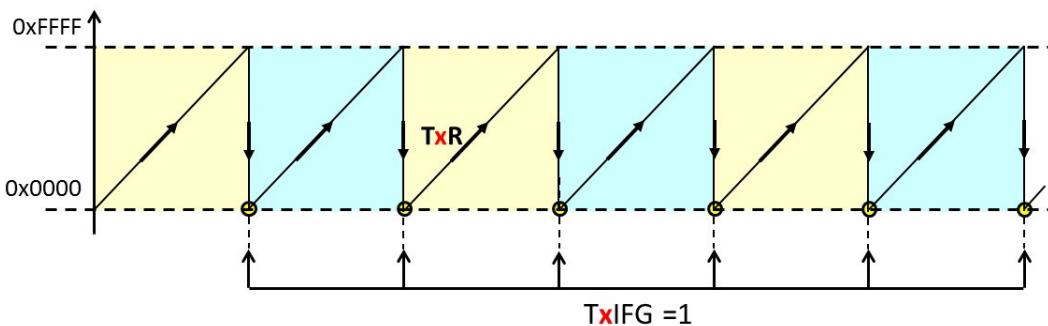


Figura 7.1. Esquema do contador de 16 bits, denominado TxR.

Por ter 16 *bits*, o contador pode ir de 0x0000 até o máximo que é 0xFFFF (65.535) e na próxima batida do relógio, ele volta a 0x0000 e recomeça a contagem. A transição do

valor máximo (0xFFFF) para o valor mínimo (0x0000) é denominada de ultrapassagem ou *overflow*. Quando isto acontece, o contador avisa fazendo o *flag* TxIFG = 1. Existe ainda o recurso de zerar o contador TxR com o uso do *bit* TxCLR. É claro que nesta zeragem, não se ativa a *flag* de *overflow*.

Todo este comportamento está representado na Figura 7.2, onde as rampas inclinadas simbolizam a contagem de TxR. As duas linhas horizontais indicam os dois limites da contagem, que são 0x0000 e 0xFFFF. O instante em que o contador volta a zero está marcado com pequenos círculos. É neste instante que a *flag* TxIFG vai para 1, como indicado.



*Figura 7.2. Gráfico indicando a contagem de TxR e os instantes em que esse contador volta a zero.*

A título de exemplo, vamos considerar TxCLK = 1 MHz, o que significa uma contagem a cada 1  $\mu$ s. Podemos então dizer que o temporizador ativa a *flag* TxIFG a cada 65.536  $\mu$ s. Note que na contagem de 0 até 65.535 existem 65.536 passos, pois o zero é também contado. É importante comentar que o contador apenas ativa a *flag* TxIFG. É obrigação do programador zerar esta *flag* para que ele possa perceber uma nova transição de 0 para 1.

Com o que foi apresentado, já podemos fazer alguma temporização. Por exemplo, se precisarmos ativar uma determinada saída ou disparar uma rotina após uma espera de 10.000  $\mu$ s, basta fazer TxR = 55.536 e esperar TxIFG ir para 1. É claro que o *hardware* oferece condições para escrever no contador TxR e zerar a *flag* TxIFG. Se, mais adiante, quisermos repetir essa espera de 10.000  $\mu$ s, precisaremos reprogramar TxR.

Podemos aperfeiçoar um pouco mais a proposta e permitir que a *flag* TxIFG provoque interrupção. Neste caso fica mais simples, pois preparamos o valor inicial de TxR e disparamos a contagem. Depois do tempo especificado, somos interrompidos pela *flag* TxIFG.

Vamos agora introduzir um comparador, que é recurso muito simples, mas que traz bastante flexibilidade ao contador. A Figura 7.3 apresenta este recurso. Note que, para beneficiar a clareza da figura, os barramentos de escrita e leitura foram omitidos. O comparador é a caixa com o sinal de igualdade “=” e para que ele opere, se faz necessário um registrador para armazenar o valor da comparação, denominado TxCCRn. Cada vez que o comparador detecta que o valor do contador TxR ficou igual (coincidiu com) ao valor do registrador TxCCRn ele avisa fazendo a flag TxCCIFGn ir para 1.

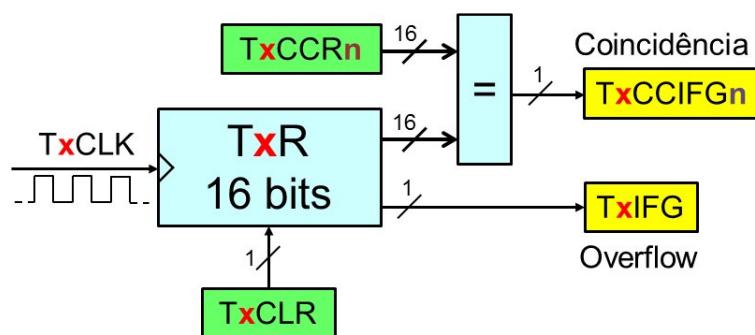
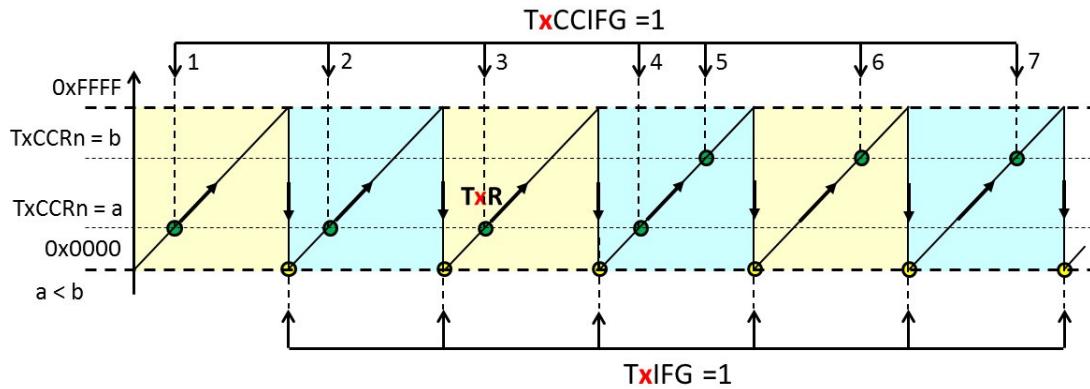


Figura 7.3. Esquema do contador de 16 bits, denominado TxR, com uma unidade de comparação.

A Figura 7.4 apresenta um gráfico que ilustra a contagem e os instantes em que o valor do contador valor ficou igual ao do registrador de comparação TxCCRn. Essas coincidências estão marcadas com um pequeno círculo e numeradas. A cada coincidência, a flag TxCCIFG vai para 1. Nesta figura, iniciamos com “TxCCR = a”, onde “a” é um valor hipotético qualquer ( $0x0000 \leq a \leq 0xFFFF$ ). No exemplo, logo após a coincidência de número 4, o programador alterou o valor do registrador, fazendo “TxCCR = b”, onde  $b > a$ . Por esse motivo, nesta rampa ascendente tivemos duas coincidências. Note que o circuito do comparador é bem simples: faz  $TxCCIFG = 1$  toda vez que tivermos  $TxR = TxCCRn$ . Sempre que o programador alterar TxCCR para um valor acima de TxR, uma coincidência deve acontecer. Entretanto, se ele alterar TxCCR para um valor abaixo de TxR, a coincidência só vai acontecer no próximo ciclo, ou seja, após TxR chegar ao máximo e voltar a zero.

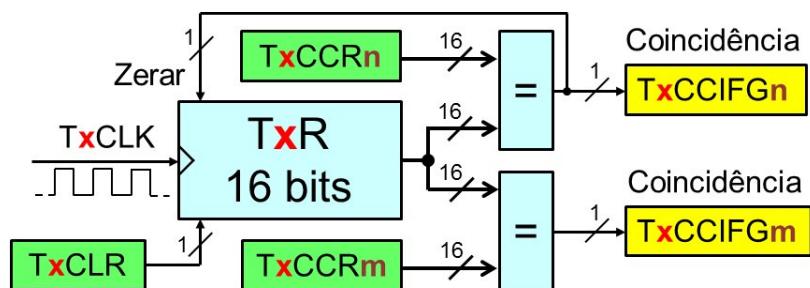


*Figura 7.4. Gráfico indicando a contagem de TxR e os instantes em que seu valor coincidiu com o do registrador de comparação TxCCRn.*

Vamos agora propor um primeiro pequeno problema. Considerando que o relógio é de 1 MHz, inicializamos: TxR = 20.000 e TxCCRn = 35.000. Uma vez iniciada a contagem, depois de quanto tempo as duas *flags* (TxIFG e TxCCIFGn) serão ativadas?

Resposta: TxCCIFGn = 1 após 15.000 µs e TxIFG = 1 após 45.536 µs.

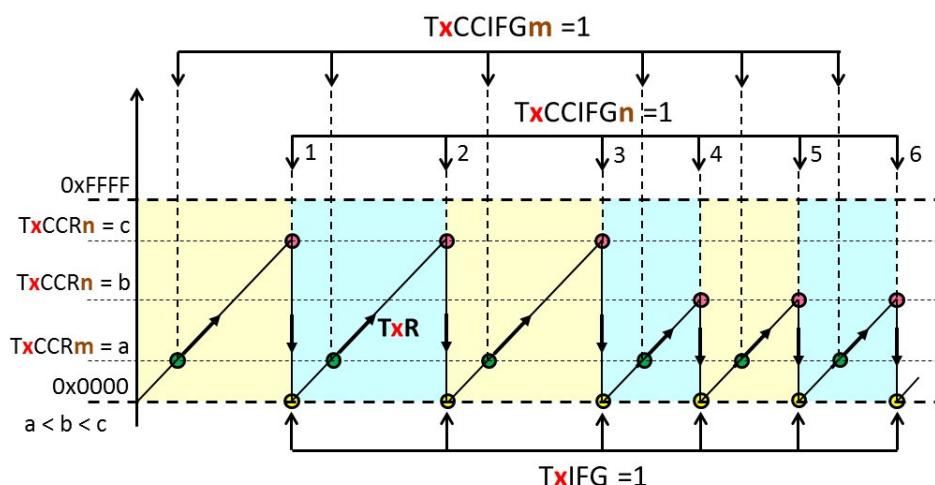
Sugerimos até agora dois recursos para atender nossas necessidades de temporização. Um inconveniente com o que foi proposto é a necessidade de se recarregar o valor inicial do contador. Pensando em eliminar essa de recarga, podemos adicionar um recurso para limitar o valor máximo de contagem. A ideia é permitir que um comparador force a zeragem de TxR quando seu valor ficar igual ao valor de um certo registrador. A Figura 7.5 traz duas unidades de comparação, são os retângulos com o sinal “=”, e dois registradores de comparação TxCCRn e TxCCRM. Quando acontece a coincidência de TxR com o valor de TxCCRn, o contador TxR é zerado. Note que somente este comparador TxCCRn tem recursos para zerar o contador. Assim, o contador sempre parte com o valor zero e vai até o limite indicado pelo registrador TxCCRn.



*Figura 7.5. Esquema do contador de 16 bits, denominado TxR, com duas unidades de comparação.*

Agora, alterando o valor de TxCCRn, podemos controlar a excursão do contador TxR, pois cada vez que TxR ficar igual a TxCCRn, ele volta a zero na próxima batida do relógio. A Figura 7.6 exibe este caso, que inicia com “TxCCRn = c” e depois da terceira coincidência, por ação do programador, teve seu valor alterado para “TxCCRn = b”. Note que esses valores “c” e “b” controlam o período da contagem. As *flags* continuam operando da mesma maneira, como listado abaixo:

- TxIFG = 1: toda vez que TxR voltar a zero;
  - TxCIFGn = 1: toda vez que  $TxR = TCCRn$  e
  - TxCIFGm = 1: toda vez que  $TxR = TCCRm$ .



*Figura 7.6. Gráfico indicando a contagem de TxR e os instantes de zeragem quando seu valor fica igual ao do registrador TxCIFGn.*

Vamos agora propor um segundo pequeno problema. Considerando que o relógio do contador TxR é de 1 MHz, e que  $TxCCRn = 4.999$  e  $TxCCRM = 3.000$ , qual a frequência de ativação de cada uma das *flags*?

Resposta: o item mais importante é o valor de TxCCRn, já que ele limita a contagem. Assim, o contador TxR sai de 0 e vai até 4.999 e, depois, volta a zero. São 5.000 contagens (lembre-se de que o zero foi contado), o que resulta em um período de 5.000  $\mu$ s. Assim, a frequência de ativação da flag TxCCIFGn é de 200 Hz. Note que as demais flags (TxIFG e TxCCIFGm) também vão ter a mesma taxa de ativação que é de 200 Hz.

Até agora, com o limitado exemplo aqui apresentado, parece que há redundância entre as flags TxIFG, TxCIFGn e TxCIFGm. Mais adiante esta impressão será removida.

Vamos adicionar mais um recurso ao *hardware* proposto. Agora será uma Unidade de Saída, cuja finalidade principal é a de gerar uma saída PWM e que está mostrada na Figura 7.7. Note que a Unidade de Saída tem a informação de quando o contador TxR voltou a zero ( $TxR = TxCCRn$ ) e de quando ele ficou igual ao valor de  $TxCCRm$ . O funcionamento desta unidade, cuja saída é OUTm, é bem simples.

- OUTm vai para 1 → sempre que  $TxR = 0$
- OUTm vai para 0 → sempre que  $TxR = TxCCRm$ .

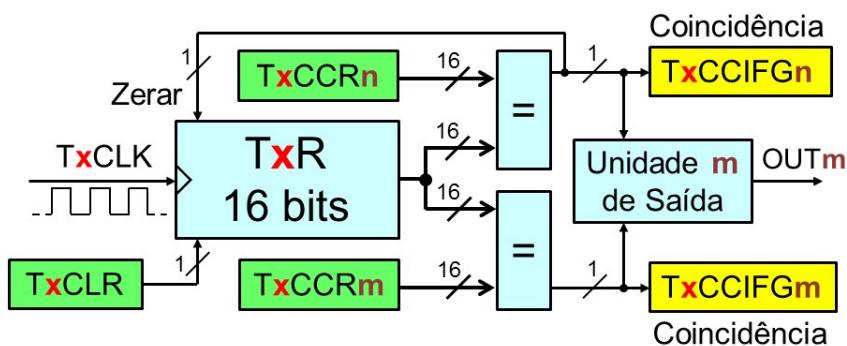
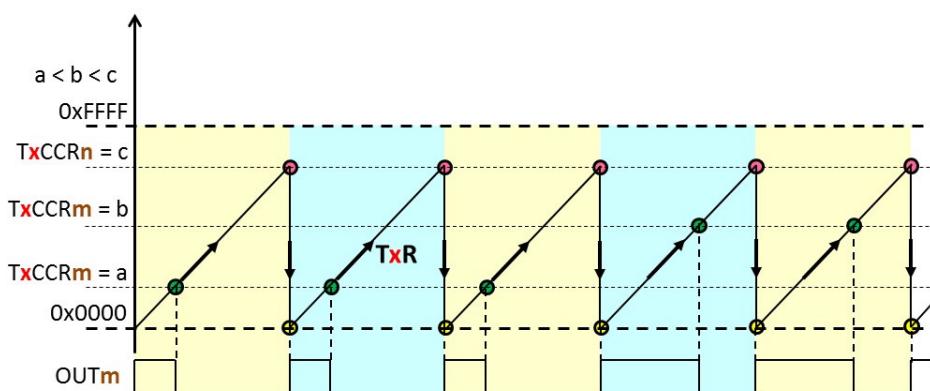


Figura 7.7. Esquema do contador de 16 bits, denominado  $TxR$ , com uma Unidade de Saída ( $OUTm$ ).

Com essa Unidade de Saída, fica muito fácil a geração de PWM. A Figura 7.8 apresenta a saída OUTm para dois valores de  $TxCCRm$ . Note que, à medida que se aumenta  $TxCCRm$ , aumenta-se o ciclo de carga. Resumindo: o valor de  $TxCCRn$  define o período do PWM e  $TxCCRm$  define seu ciclo de carga, de acordo com a expressão abaixo.

$$\text{Ciclo de Carga (\%)} = \frac{TxCCRm}{TxCCRn} \times 100\%$$



*Figura 7.8. Gráfico indicando a geração de uma saída PWM com os registradores TxCCRn e TXCCRM.*

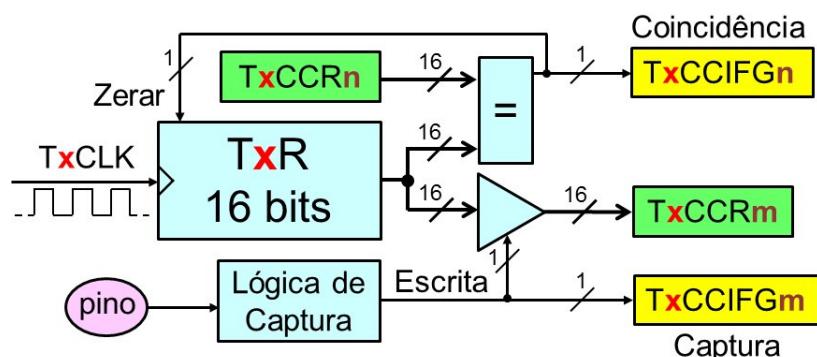
Vamos agora propor um terceiro pequeno problema. Considerando que o relógio do contador TxR é de 1 MHz, indique os valores de TxCCRn e TXCCRM para gerar uma saída PWM com período de 10 ms e ciclos de carga de 30% e 70%.

Resposta: o período é controlado pelo valor de TxCCRn e o ciclo de carga pelo valor de TXCCRM. As contas são simples:

- Para o período de 10 ms, precisamos de 10.000 contagens ( $1.000.000 \times 0,01$ ) e, com isso, temos TxCCRn = 9.999, pois o zero é também contado.
- Para um ciclo de carga de 30%, temos TXCCRM =  $0,3 \times 9.999 = 2.999,7$ , que aproximamos para 3.000 contagens.
- Para um ciclo de carga de 70%, temos TXCCRM =  $0,7 \times 9.999 = 6.999,3$ , que aproximamos para 7.000 contagens.

Finalmente, outro recurso interessante a ser adicionado ao temporizador proposto é a possibilidade de captura de eventos. Por capturar um evento, simplesmente se quer dizer marcar o instante de tempo em que aconteceu uma alteração num determinado pino, que pode ser um flanco de subida ( $\uparrow$ ) ou um flanco de descida ( $\downarrow$ ), ou ainda, qualquer um deles. A Figura 7.9 apresenta um diagrama com esse novo recurso. Na parte inferior da figura, está a Lógica de Captura, onde o usuário indica qual flanco deseja capturar. Quando acontece o evento selecionado, a Lógica de Captura copia o valor do contador TxR para o registrador TXCCRM e, marca esse fato com TXCCIFGm = 1.

É de se notar que o registrador TXCCRM, que nas figuras anteriores era usado pelo comparador, passou a ser dedicado à captura de eventos. Não está mostrado nesta proposta, mas é claro que o programador deve poder indicar se quer operar no modo captura ou no modo comparação.



*Figura 7.9. Esquema do contador de 16 bits, com uma lógica de captura.*

Para explicar ainda mais, vamos supor que se deseja medir quanto tempo um determinado sinal digital fica em nível alto. Para resolver esse problema, ligamos este sinal ao “pino” indicado na figura acima, selecionamos o modo de captura e preparamos a Lógica de Captura para o evento flanco de subida. Quando ele acontecer, o valor de TxR é copiado para o registrador TCCRm. O programa percebe que aconteceu uma captura porque a *flag* TCCIFGm vai para 1. Então, ele copia o valor de TCCRm para uma variável qualquer, zera a *flag* correspondente e programa a Lógica de Captura para o evento flanco de descida. Quando acontecer esse segundo evento, a diferença entre o valor atual TCCRm e o valor que foi guardado anteriormente vai indicar (em contagens de TxR) quanto tempo o sinal ficou em nível alto.

Uma dúvida comum sobre a captura é: ao invés de usar o *hardware* descrito, não posso escrever um programa para fazer tudo isso? Sim, mas um programa vai introduzir erro, pois consome tempo para detectar a alteração no pino e copiar o valor de TxR para uma variável. Outra pergunta: será que não poderia monitorar a alteração do pino com o uso de uma interrupção? Esta solução é pior ainda, porque as interrupções possuem uma latência grande. Além disso, se o evento de captura acontecer enquanto se atende a uma outra interrupção, o atraso será maior ainda. Por outro lado, o *hardware* que foi apresentado é capaz de copiar o valor de TxR para o registrador TCCRm com a precisão de um período de relógio do contador.

Temos agora uma boa ideia do que esperar de um Temporizador/Contador. Ele deve oferecer ao programador recursos para:

- Ativar *flags* com a periodicidade desejada;
- Provocar interrupções periódicas;
- Gerar atrasos de tempo;
- Controlar saída PWM e
- Medir intervalo entre dois eventos.

Passemos então ao estudo da arquitetura dos Temporizadores/Contadores disponíveis na família MSP430F5529.

## 7.2. Introdução aos *Timers* do MSP430F559

Iniciamos explicando a forma como o fabricante identificou esses recursos. A arquitetura MSP430 oferece três diferentes tipos de *timers*:

- Timer A;
- Timer B e
- Timer D.

Cada versão da família MSP430 pode ter diferentes quantidades de um determinado tipo de *timer*, essas versões são chamadas de instâncias. No nosso caso, o MSP430F5529 tem 3 instâncias do Timer A, denominadas de TA0, TA1 e TA2, apenas uma instância do Timer B, denominada de TB0 e nenhuma instância do Timer D. Cada instância possui um contador e vários registradores de Comparação e Captura, formando a sigla “**CC**”, que aparece no nome dos registradores.

*Tabela 7.1. Instâncias dos timers disponibilizados pelo MSP430F5529*

-	<b>Timer A0</b>	<b>Timer A1</b>	<b>Timer A2</b>	<b>Timer B0</b>
Contador	TA0R	TA1R	TA0R	TB0R
Registrador Compara/Captura	TA0CCR0	TA1CCR0	TA2CCR0	TB0CCR0
Registrador Compara/Captura	TA0CCR1	TA1CCR1	TA2CCR1	TB0CCR1
Registrador Compara/Captura	TA0CCR2	TA1CCR2	TA2CCR2	TB0CCR2
Registrador Compara/Captura	TA0CCR3	-	-	TB0CCR3
Registrador Compara/Captura	TA0CCR4	-	-	TB0CCR4
Registrador Compara/Captura	-	-	-	TB0CCR5
Registrador Compara/Captura	-	-	-	TB0CCR6

Para tornar a ideia bem clara, apresentamos a Figura 7.10 onde se vê a instância zero do Timer A (TA0), composto por seu contador (TA0R) e pelos 5 registradores de comparação e captura (TA0CCR0, TA0CCR1, ..., TA0CCR4). O leitor deve começar a prestar atenção aos nomes dos registradores e ir se acostumando com sua lógica de formação.

Existe um registrador para o controle do contador e um registrador para o controle de cada unidade de comparação. Por exemplo, o controle do contador TA0R é feito pelo registrador TA0CTL. O controle de cada uma das unidades de comparação de TA0 é feito pelos registradores TA0CCTL0, TA0CCTL1, ..., TA0CCTL4.

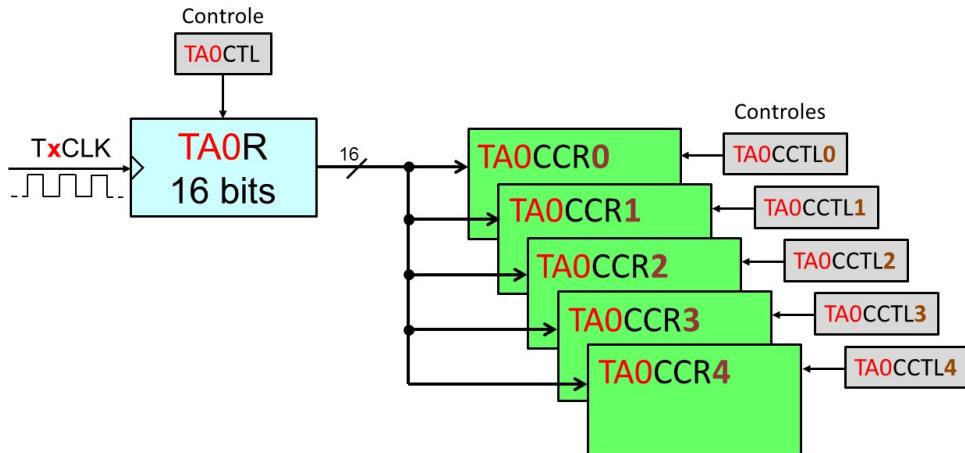


Figura 7.10. Instância zero do Timer A (TA0R), com seus 5 registradores de comparação e captura (TA0CCR0, TA0CCR1, ..., TA0CCR5) e os registradores de controle.

Faremos a seguir o estudo do Timer A e de suas instâncias. Depois estudaremos o Timer B vendo suas diferenças e particularidades em relação ao Timer A.

Tabela 7.2. Distribuição das entradas de captura nas Portas GPIO para a arquitetura F5529 (em cinza estão os pinos não disponíveis no Launch Pad)

Timer A0		Timer A1		Timer A2		Timer B0	
TA0.0	P1.1	TA1.0	P1.7	TA1.0	P1.7	TB0.0	P5.6
TA0.1	P1.2	TA1.1	P2.0	TA1.1	P2.0	TB0.1	P5.7
TA0.2	P1.3	TA1.2	P2.1	TA1.2	P2.1	TB0.2	P7.4
TA0.3	P1.4					TB0.3	P7.5
TA0.4	P1.5					TB0.4	P7.6
						TB0.5	P3.5
						TB0.6	P3.6

### 7.3. Estudo do Timer A

Como já vimos, o MSP430F5529 oferece 3 instâncias do *Timer A*, cada uma com certa quantidade de comparadores (vide Tabelas 7.1 e 7.2). Para facilitar a explicação e as referências adotaremos as seguintes convenções para denominar os registradores pertinentes.

- A letra “x” indica uma instância do *timer A* ( $x = 0, 1, \dots$ ) e
- A letra “n” indica uma instância da unidade captura e comparação ( $n = 0, 1, \dots$ ).

Como já foi afirmado e está ilustrado na Figura 7.10, existem registradores para controlar a operação do contador e o funcionamento das unidades de captura e comparação. A tabela abaixo resume esses registradores. Cada instância de um *timer* possui um registrador de controle denominado TAxCTL e os diversos registradores de controle das unidades de captura/compara deste *timer* são denominados TAxCCTLn.

*Tabela 7.3. Resumo dos registradores das instâncias do Timer A e de seus registradores de Captura/Compara*

Finalidade	Registrador	Controle	Flag IFG
Contador	TAxR	TAxCTL	TAIFG
Captura/Compara	TAxCCRn	TAxCCTLn	CCIFG

A lógica usada na nomenclatura dos registradores é importante e sua compreensão facilita muito o uso dos *timers*. Assim, insistimos no tema e apresentamos um resumo:

- TAxR → contador da instância **x** do *timer* A;
- TAxCTL → registrador para controle do contador da instância **x** do *timer* A;
- TAxCCRn → registrador da instância **n** do módulo de captura/compara de TAx e
- TAxCCTLn → registrador de controle da instância **n** do módulo de captura/compara de TAx.

É importante indicar que cada registrador de controle tem uma *flag* para indicar o progresso de seu contador. Nos casos em que essas *flags* têm o mesmo nome, a distinção entre elas é feita pelo registrador de controle onde ela se encontra. Na lista abaixo usamos a notação “registrador.*flag*”.

- TAxCTL.TAIFG → indica que TAxR voltou a zero e
- TAxCCTLn.CCIFG → indica que o valor de TAxR coincidiu com o de TAxCCR0.

### 7.3.1. Contagem com o *Timer A*

Apesar de ser óbvio, iniciamos afirmando que o uso mais simples de um *timer* é para a contagem. O *Timer A* possui 4 modos de contagem:

- Modo 0 → Parado (*stop*);
- Modo 1 → Ascendente (*up*);
- Modo 2 → Contínuo (*continuous*) e
- Modo 3 → Ascendente/Descendente (*up/down*).

#### 7.3.1.1. Modo 0 – Parado (*Stop*)

Quando no Modo 0 (Parado ou *Stop*), o contador fica parado. Então este modo serve apenas para parar a contagem de TAxR.

### 7.3.1.2. Modo 1 – Ascendente (*up*)

Quando no Modo 1 - Ascendente (*up*), o contador TAxR conta até seu valor ficar igual ao do registrador TAxCCTLn, quando então ele recomeça a contagem a partir de zero, como mostrado na Figura 7.11. Este modo permite controlar a excursão do contador. O período será igual a TAxCCTLn+1 contagens, porque o zero é também contado. Se este modo é selecionado com TAxR maior que TAxCCTLn, o contador é zerado imediatamente. Note que somente o comparador 0 (TAxCCTL0) pode ser usado para zerar o TAxR.

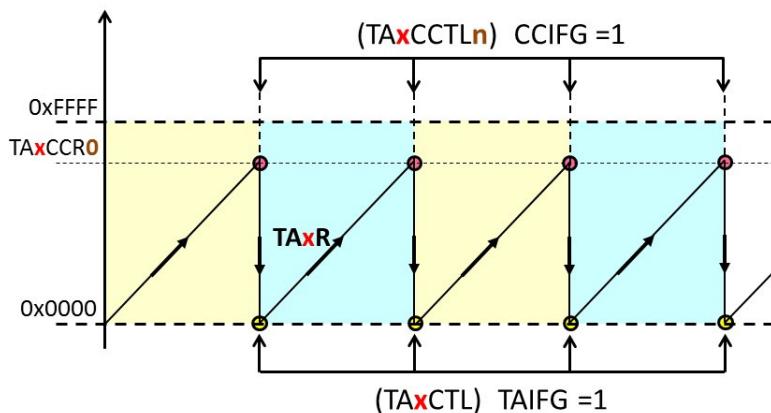


Figura 7.11. Modo 1 - Ascendente (*up*), o contador TAxR é zerado quando seu valor fica igual ao do registrador TAxCCTL0.

A Figura 7.11 ainda mostra o instante de ativação das *flags* CCIFG e TAIFG. Quando o contador TAxR fica igual ao registrador de comparação TAxCCTL0 a *flag* CCIFG é ativada (além de zerar TAxR) e quando TAxR fica igual a zero, a *flag* TAIFG é ativada. Essas *flags* estão em diferentes registradores de controle, como indicado na Figura 7.11 (note o nome entre os parêntesis). Para melhor visualização dessas *flags*, consulte as figuras 7.24 e 7.26.b. O resumo abaixo repete o conceito, usando a notação “*registraror.flag*”:

- TAxCTL.TAIFG → indica que TAxR voltou a zero e
- TAxCCTL0.CCIFG → indica que o valor de TAxR coincidiu com o de TAxCCTL0.

O instante exato em que as *flags* são ativadas é importante e está mostrado na Figura 7.12. Note que na batida do relógio que faz TAxR igual a TAxCCTL0, a *flag* CCIFG vai para 1. Já na batida que leva TAxR para zero, a *flag* TAIFG vai para 1.

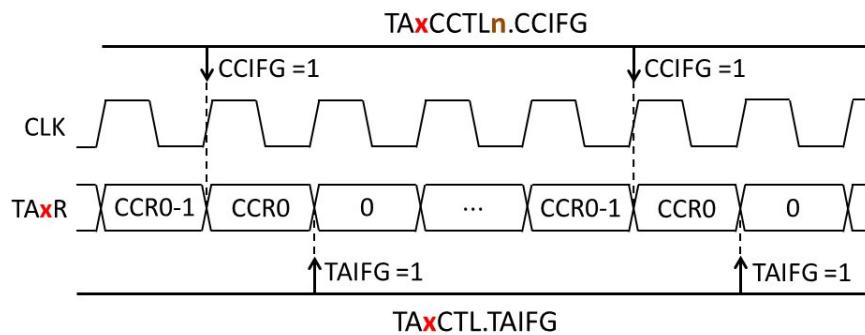


Figura 7.12. Modo 1 (Up ou Ascendente), mostrando o instante em que cada flag é ativada.

Para tornar o conceito ainda mais claro, a Figura 7.13 apresenta o exemplo de se empregar o Modo 1 com  $TAxCCR0 = 999$ . Serão 1.000 contagens. Cada vez que o contador  $TAxR$  fica igual a  $TAxCCR0$  (que é 999), a flag  $CCIFG$  vai para 1. Na próxima batida do relógio o contador  $TAxR$  volta a zero e a flag  $TAIFG$  vai para 1.

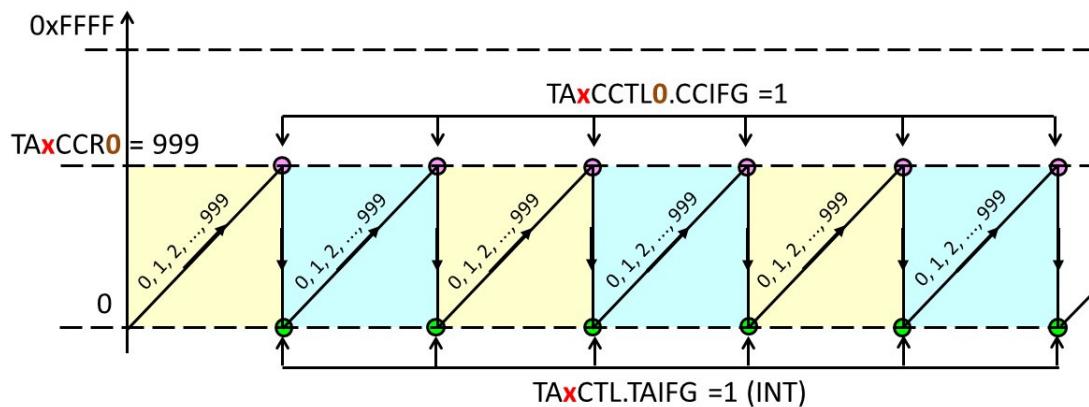


Figura 7.13. Modo 1 - Ascendente (Up), usando exemplo de  $TAxCCR0 = 999$  e mostrando o instante em que cada flag é ativada.

O valor do contador  $TAxR$  pode ser lido ou escrito a qualquer momento. **Se o novo valor escrito em  $TAxR$  for maior que o valor de  $TAxCCR0$ ,  $TAxR$  é zerado automaticamente, ou seja, a contagem segue a partir do zero (verificar).** Se o valor do comparador  $TAxCCR0$  for alterado para um valor idêntico ou superior ao atual, tudo se passa como o esperado. Porém, se  $TAxCCR0$  for alterado para um valor abaixo do valor atual de  $TAxR$ , este registrador  $TAxR$  é zerado. Entretanto, o manual alerta que uma contagem extra pode ocorrer.

### 7.3.1.3. Modo 2 – Contínuo (*Continuous*)

Quando no Modo 2 – Contínuo (*Continuous*), o contador TAxR conta até seu limite de 16 bits, que é o valor 0xFFFF. Quando chega a este limite, o contador volta a zero e a flag TAIFG vai para 1. Note que nenhum registrador de comparação é usado para limitar a contagem. A Figura 7.14 ilustra este modo de operação, que é semelhante ao modo anterior.

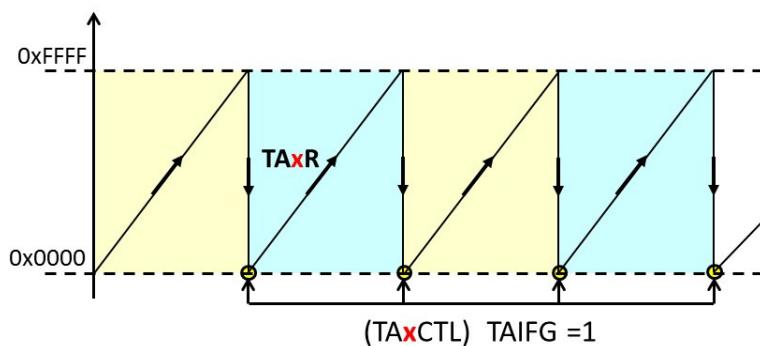


Figura 7.14. Modo 2 - Contínuo (*Continuous*), o contador conta até seu limite de 16 bits (0xFFFF) e em seguida volta a zero. Cada vez que volta a zero, a flag TAIFG vai para 1.

### 7.3.1.4. Modo 3 – Sobe/Desce (Up/Down)

Quando no Modo 3 – Sobe/Desce (*Up/Down*), o contador TAxR inicia contando de forma ascendente e conta até ficar igual ao valor de TAxCCR0, quando então passa a contar de forma descendente até voltar a zero e reiniciar o ciclo. O contador TAxR fica um período com valor igual ao de TAxCCR0 e um período em zero. A Figura 7.15 ilustra este modo de operação. É importante notar o instante de ativação das flags TAIFG e TAxCCIFG0. A direção da contagem é armazenada em um *latch*, isso permite que o contador continue no mesmo sentido, caso ele seja parado momentaneamente. Caso o contador seja zerado usando o TACLR, a contagem inicia a partir do zero e em modo ascendente (o bit TACLR será estudado mais adiante).

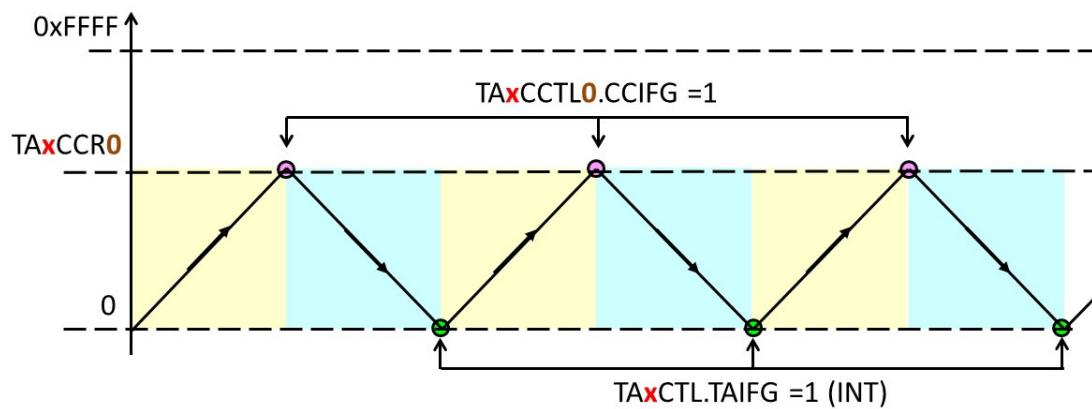


Figura 7.15. Modo 3 - Sobe/Desce (Up/down), o contador conta até seu limite de 16 Bits (0xFFFF) e em seguida volta a zero. Cada vez que volta a zero, a flag TAIFG vai para 1.

O valor do contador TAxR pode ser lido ou escrito a qualquer momento. **Se o novo valor escrito em TAxR for maior que o valor do comparador TAxCCR0, este registrador TAxR é zerado, ou seja, a contagem segue a partir do zero (verificar).** Se o valor de TAxCCR0 for alterado durante a fase descendente, o contador TAxR segue normalmente até chegar ao zero e somente na fase ascendente é que o novo valor de TAxCCR0 será efetivado. Se TAxCCR0 for alterado durante a fase ascendente para um valor igual ou superior ao atual, a contagem segue da forma esperada. Se TAxCCR0 for alterado durante a fase ascendente para um valor inferior ao valor atual de TAxR, a contagem recomeça de forma ascendente a partir do zero.

### 7.3.2. Diagrama de Blocos do Contador

A Figura 7.16 apresenta um diagrama de blocos do contador. De acordo com a convenção adotada, todos os retângulos de cantos arredondados (em verde) indicam *bits* que são configurados (controlados) pelo programador. Os retângulos preenchidos com amarelo vivo indicam *flags* que o programador pode consultar ou usar para provocar interrupção. Essas *flags* têm os nomes de acordo com o padrão “xxxIFG”. Os pinos do processador são indicados por nomes dentro de formas circulares.

Nesta figura é muito importante a seleção do relógio (CLK) do contador TAxR. Como se vê no extremo esquerdo, graças a um multiplexador, o programador pode escolher uma dentre 4 fontes distintas. É possível usar um dos relógios internos, ACLK ou SMCLK, ou o usuário pode fornecer um relógio (TAxCLK) por um pino específico do processador. A entrada de número 3 (INCLK) não está disponível na versão F5529. Depois de selecionado o relógio desejado, ele pode ser submetida a dois divisores cascataeados até se transformar em CLK, que é o relógio do contador TAxR.

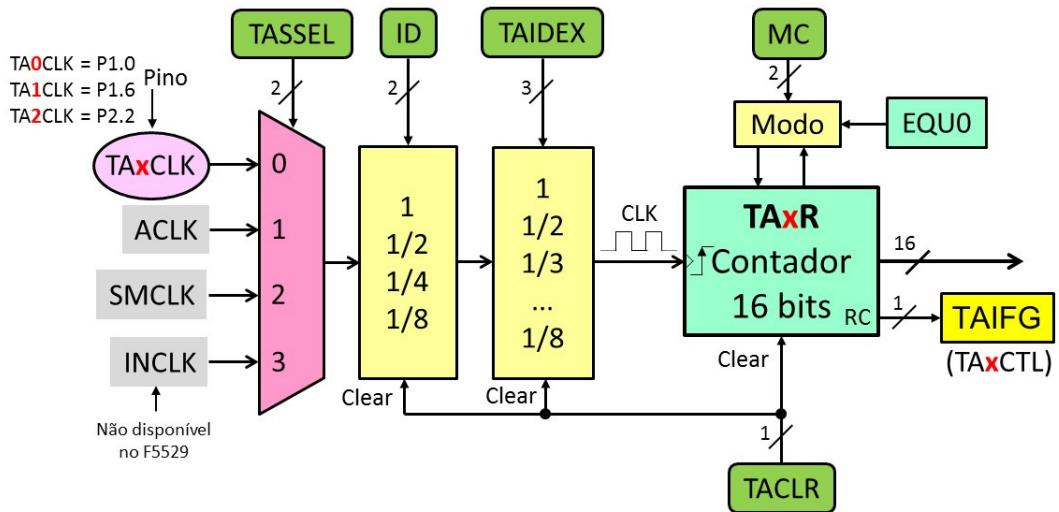
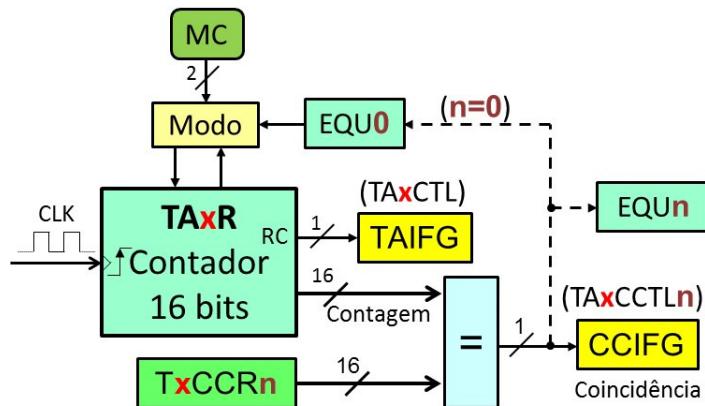


Figura 7.16. Diagrama de blocos para os modos de contagem do Timer A (TAXR). Nota-se de importante a seleção do relógio (CLK) para a contagem.

De acordo com o modo selecionado (MC), o contador TAxR conta na frequência do sinal CLK selecionado. Cada vez que esse contador volta a zero, a flag TAIFG é ativada (pela saída RC, que em inglês significa *Ripple Carry Out*). Essa flag TAIFG não tem nome próprio, ela é individualizada pelo registrador de controle ao qual pertence (TAXCTL). A caixa denominada “EQU0”, indica o instante em que o contador TAxR ficou igual ao registrador TAxCCR0, pois como já foi visto, isto é importante para os modos 1 e 3. Para finalizar, o bit “TACLR” faz a zeragem do contador TAxR e dos estados internos dos dois divisores. Um ponto interessante é que o usuário escreve 1 neste bit e, após a zeragem do contador, ele volta automaticamente para 0.

A Figura 7.17 apresenta apenas a parte ligada à comparação. O comparador é a caixa marcada com o sinal de igualdade “=”. Ele, continuamente, compara o valor do contador TAxR com o valor que o programador armazenou no registrador de comparação TAxCCRn. A flag CCIFG vai a 1 cada vez que esses dois registradores têm o mesmo valor, fato designado como “coincidência”. Note que essa flag CCIFG não tem nome próprio, ela é individualizada pelo registrador de controle do comparador ao qual pertence (TAXCCTLn). A Tabela 7.1, já apresentada, lista a quantidade de comparadores para cada instância do timer A.



*Figura 7.17. Diagrama de blocos para os modos de contagem, apresentando um dos registradores de comparação ( $n = 0, 1, \dots, 5$ )*

Quando ocorre a coincidência entre o valor do contador TAxR e do comparador TxCCRn, um sinal interno EQUn é ativado. Esse sinal é importante na geração de saídas, como será visto mais adiante. Por enquanto, o que nos interessa é o EQU0, pois ele é o responsável por zerar o TAxR quando este opera nos modos 1 ou 3.

**Exemplo 7.1:** Usando o *Timer TA0* com ACLK = 32.768 Hz, construa um relógio de 24 horas com precisão de décimos de segundos.

**Solução:** Vamos usar o Modo 1 (Ascendente), onde o limite de contagem é indicado pelo valor de TA0CCR0. O ACLK faz 32.768 contagens em 1 segundo, portanto, em 0,1 s (um décimo) são 3.276,8 contagens. Como precisamos trabalhar com inteiros, arredondamos para 3.277 contagens e comentemos um pequeno erro (0,610 µs). O relógio vai atrasar um pouco. Lembrando que nessa contagem, o zero é também contado, subtraímos 1 desse valor, que passa a ser 3.276. No programa listado a seguir, são usados alguns registradores, que serão completamente descritos mais adiante.

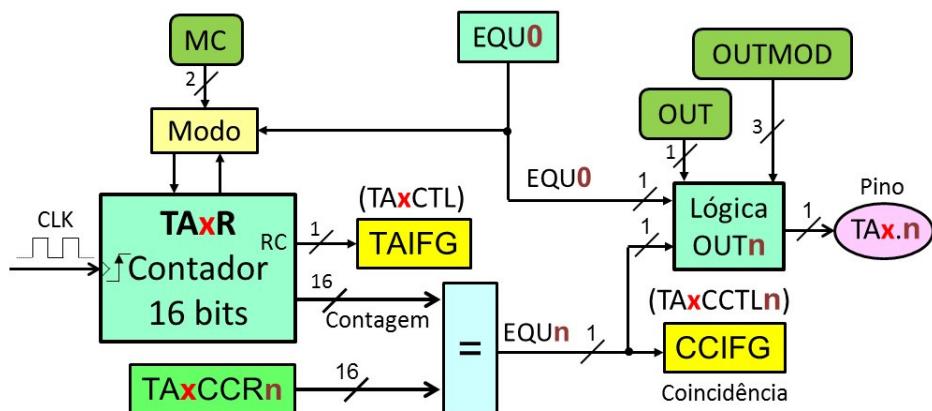
```
# define TRUE 1
void relógio (void){
unsigned char hora=0, min=0, seg=0, dseg=0;
TA0CTL = TASSEL_1 | MC_1 | TACL; //Selecionar ACLK e Modo 1
TA0CCR0 = 3277; //Limite da contagem
while(TRUE){
    while ( (TA0CTL&TAIFG) == 0); //Esperar TAIFG=1;
    TA0CTL &= ~TAIFG; //Fazer TAIFG=0
    if (++dseg == 10){ //Incr. décimos de segundos
        dseg=0;
        if (++seg == 60){ //Incr. segundos
            seg=0;
            if (++min == 60){ //Incr. minutos
                min=0;
                hora++;
            }
        }
    }
}
```

```
        min=0;
        if (++hora == 24) //Incr. horas
            hora=0;
    }
}
}
```

### 7.3.3. Geração de Saída com *Timers*

Como já vimos, são várias instâncias dos *timers*, sendo que cada instância tem vários comparadores. Cada comparador possui uma Unidade Geradora de Saída. Essa unidade, usando os sinais (EQUn) fornecidos pelos comparadores, pode gerar diversos tipos de saída, para cada modo de contagem.

A Figura 7.18 apresenta o diagrama de blocos para o estudo da Unidade Geradora de Saída do comparador n, que na figura foi rotulada com “Lógica OUTn”. Essa Lógica OUTn usa as informações (EQU0 e EQUn) fornecidas pelas Unidades de Comparação. De acordo com o modo de saída programado (OUTMOD) um determinado sinal é gerado no pino do processador. É claro que o modo de operação do contador TAxR vai influenciar diretamente na forma do sinal gerado.



*Figura 7.18. Diagrama de blocos para a Unidade Geradora de Saídas (Lógica OUTn) do comparador n ( $n = 0, 1, \dots, 5$ ).*

A Tabela 7.4 relaciona o comportamento da saída em função do Modo de Operação da Unidade de Saída (OUTMOD) e das coincidências (EQU0 e EQU1) sinalizadas pelas unidades de comparação. A palavra inglesa “*Toggle*”, usada nesta tabela, significa inversão do estado da saída. O Modo de Saída 0, permite que a saída seja controlada

diretamente pelo *bit* OUT do registrador de controle (TAxCCRn). Em outras palavras, neste modo (OUTMOD = 0), a saída acompanha o *bit* OUT, que é controlado pelo programador. Os modos 2, 3, 6 e 7 usam sinais de duas unidades de comparação sendo que, forçosamente, a unidade 0 é uma delas. Assim, esses modos não podem ser utilizados com a unidade de saída do comparador 0. As Figuras 7.19, 7.20 e 7.21 ilustram esses diversos modos de operação da Unidade de Saída

*Tabela 7.4. Apresenta os 8 modos de operação da Unidade Geradora de Saída.*

<b>OUTMOD</b>	<b>Modo</b>	<b>EQUn</b>	<b>EQU0</b>
0	<i>Output</i>	Output	Output
1	<i>Set</i>	1	-
2	<i>Toggle/Reset</i>	T	0
3	<i>Set/Reset</i>	1	0
4	<i>Toggle</i>	T	-
5	<i>Reset</i>	0	-
6	<i>Toggle/Set</i>	T	1
7	<i>Reset/Set</i>	0	1

T = Inverte estado da saída

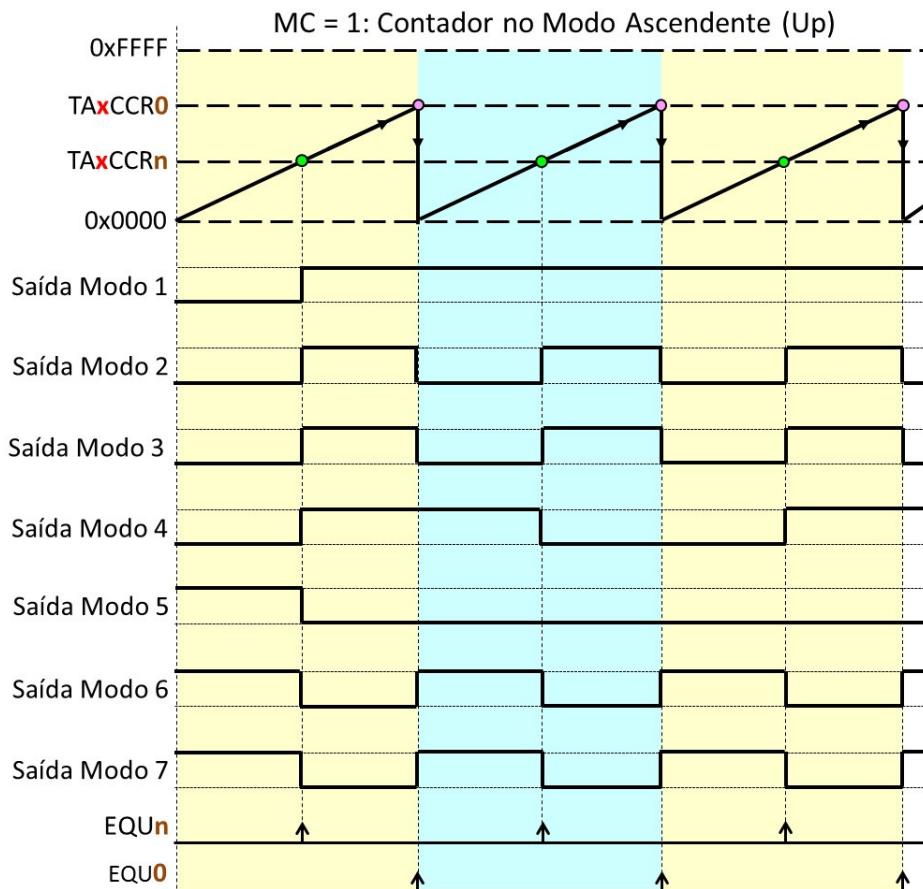
Espaço em branco intencional.

A Figura 7.19 apresenta os Modos de Saída do Comparador n quando o contador está operando no Modo Ascendente (*Up*, MC = 1). Neste Modo Ascendente, o registrador TAxCCR0 é usado para limitar a excursão da contagem, ou seja, ele especifica o período das formas de onda geradas na saída. A saída n (OUTn) é controlada pelo Comparador n. Note que alguma coisa acontece com a saída quando o contador TAxR fica igual ao valor de TAxCCRn. Este Modo Ascendente é o mais indicado para gerar PWM de alta frequência, como é o caso de regulação de potência, de retificação e de construção de conversores Digital/Analógico (DAC).

Como se pode ver na figura, os modos 2, 3, 6 e 7 são os mais indicados para a geração de PWM. Neste caso, temos:

- TAxCCR0 → especifica o período do PWM e
- TAxCCRn → especifica o ciclo de carga do PWM.

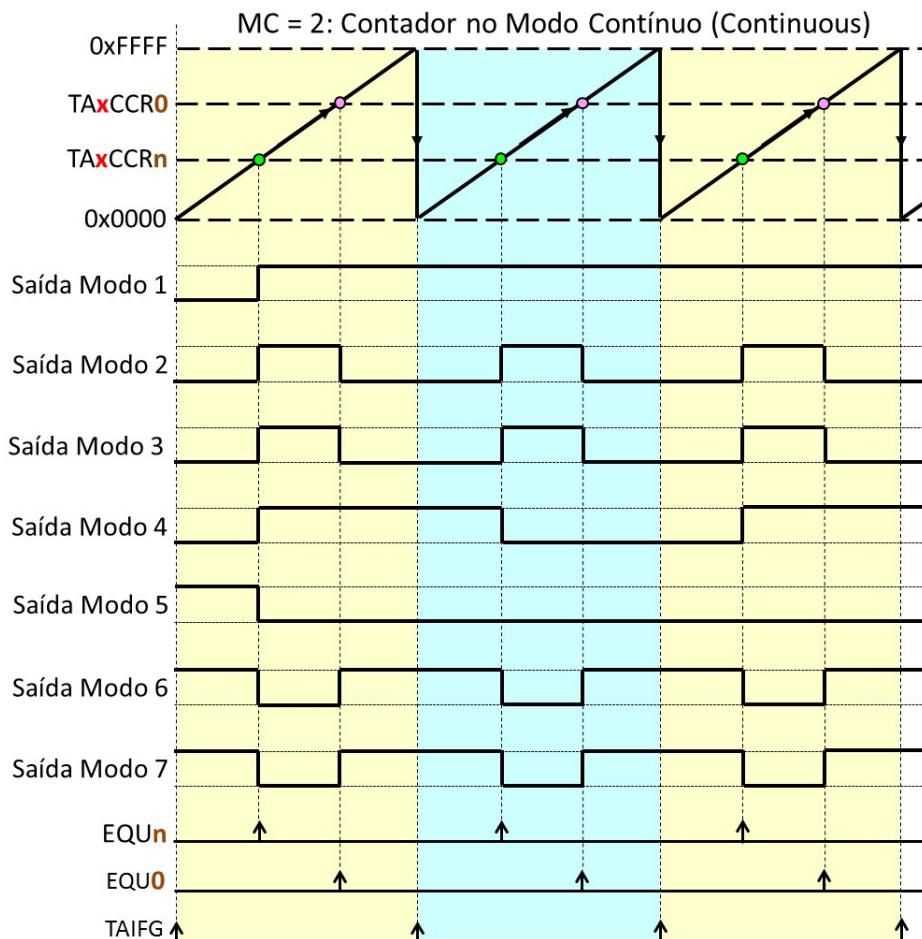
Não está indicado na Figura 7.19, mas as *flags* CCIFG continuam sendo ativadas cada vez que o valor do contador TAxR coincide com o valor de algum registrador de comparação (TAxCCRn).



*Figura 7.19. Opções de Modos de Saída para quando o contador opera no Modo Ascendente.*

A Figura 7.20 apresenta os Modos de Saída do Comparador n quando o contador está operando no Modo Contínuo (*Continuous*, MC = 2). Neste modo, o limite de contagem é fixo e igual a 0xFFFF e, portanto, o período dos sinais gerados também é fixo. A geração da saída n (OUTn) envolve o registrador de comparação TAxCRn e, forçosamente, o registrador TAxCCR0. Note que alguma coisa acontece com a saída quando o contador TAxR fica igual ao valor de um dos registradores TAxCCR0 ou TAxCRn. Este modo de contagem permite que selecione o instante exato dos flancos de subida e de descida da saída gerada, ou seja, permite que se posicione a carga do PWM em qualquer lugar dentro do seu período. Este modo já é mais indicado para o controle de motores. **Quais outras aplicações?**

Não está indicado na Figura 7.20, mas as *flags* CCIFG continuam sendo ativadas cada vez que o valor do contador TAxR coincide com o valor de algum registrador de comparação (TAxCRn).

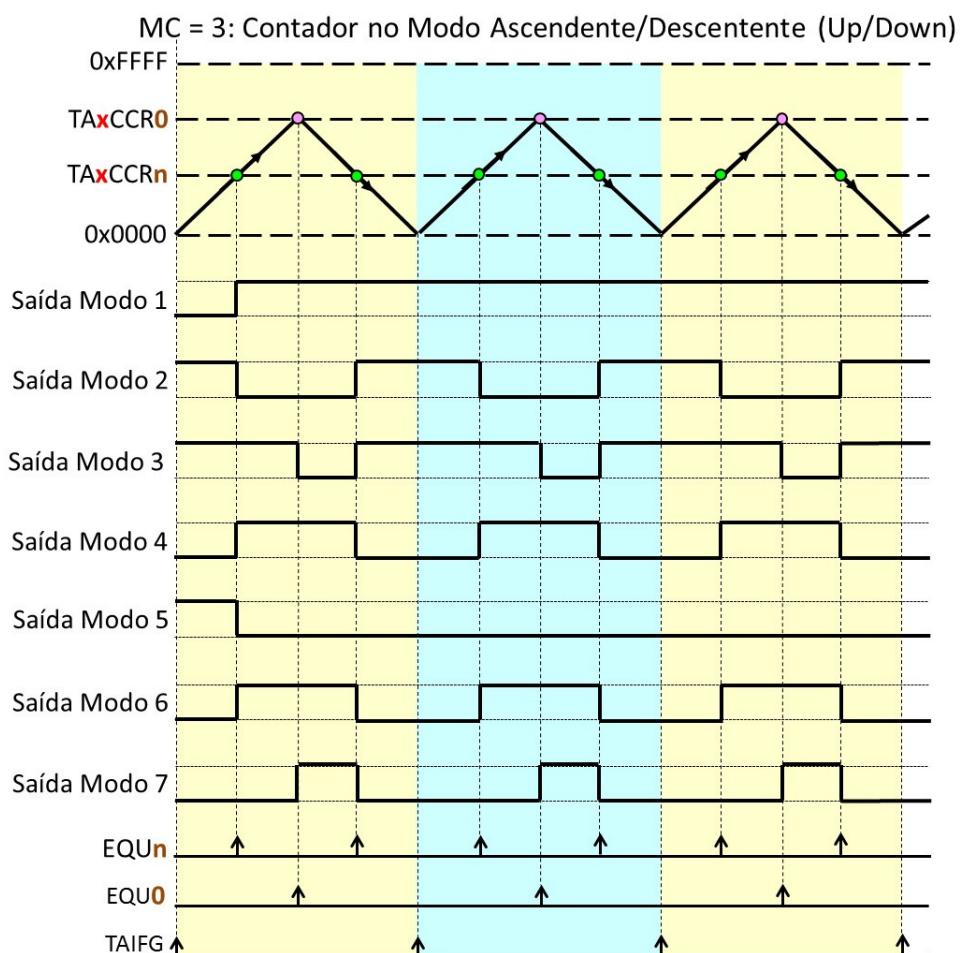


*Figura 7.20. Opções de Modos de Saída para quando o contador opera no Modo Contínuo.*

A Figura 7.21 apresenta os Modos de Saída do Comparador n quando o contador está operando no Modo Sobe/Desce (*Up/down*, MC = 3). Neste modo, o limite de contagem é especificado pelo valor de TAxCCR0 e o contador TAxR vai de forma ascendente desde o zero até valor de TAxCCR0, quando então ele passa a contar de forma descendente até chegar a zero e recomeçar tudo de novo.

Como se pode ver na figura, os modos 2, 3, 4, 6 e 7 são os mais indicados para a geração de PWM. Neste caso, temos:

- TAxCCR0 → especifica (metade do) o período do PWM e
- TAxCCRn → especifica o ciclo de carga do PWM.



*Figura 7.21. Opções de Modos de Saída para quando o contador opera no Modo Ascendente/Descendente (MC=3).*

Este modo é o mais recomendado para a geração de PWM para o controle de motores, especialmente quando se precisa alterar o ciclo de carga com grande frequência. Note que nos modos 2, 4 e 6 o centro do pulso gerado coincide com o pico da contagem. Isto

garante que os pulsos sempre tenham o mesmo espaçamento temporal, independente do ciclo de carga. Em outros modos de geração de saída, uma modificação no ciclo de carga do PWM altera o intervalo entre dois pulsos consecutivos. A Figura 7.20 ilustra esse fato comparando os modos de contagem 1 e 3. Note que na parte superior (Modo 1) desta figura, o espaçamento entre os pulsos está muito irregular, a pior parte está marcada com uma elipse colorida. Já na parte inferior (Modo 3), há uma melhor distribuição, pois o centro do pulso está alinhado com o instante de máxima contagem.

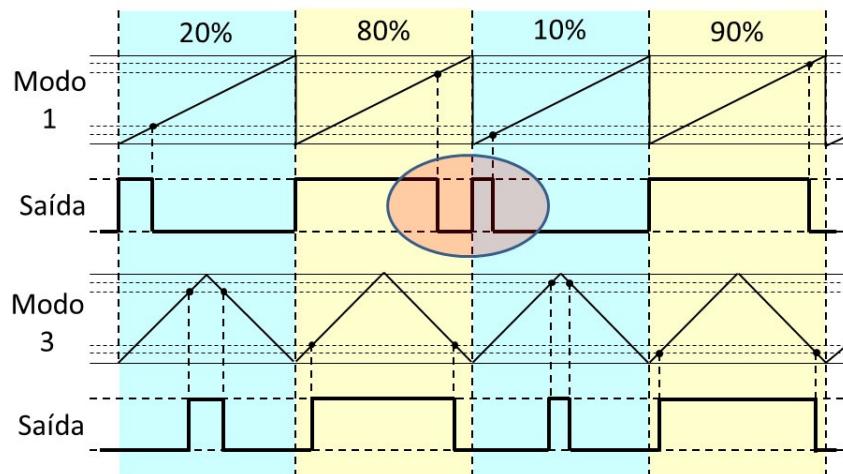


Figura 7.22. Comparação na geração de saída com o contador operando em Modo 1 (Up) e em Modo 3 (Up/Down).

### 7.3.4. Captura de Eventos com *Timers*

Como já vimos no início deste capítulo, a captura nada mais é que a cópia do contador TAxR para um registrador de comparação TxCCRn quando o evento especificado acontece. O evento pode ser, por exemplo, um determinado pino indo para nível alto. Isto permite que se meça o intervalo de tempo entre dois eventos. Serve para medir largura de pulsos, periodicidade de um determinado sinal etc.

A Figura 7.23 apresenta o diagrama de blocos da unidade de captura “n”. O multiplexador mais à esquerda, denominado de MUX1, controlado pelos bits CCIS seleciona a entrada que será usada para gerar o evento de captura. São 4 entradas possíveis. Entretanto, nesta versão F5529, somente a entrada CCInA está conectada a um pino do processador. A Tabela 7.6 apresenta uma lista dos pinos que nesta versão do MSP podem disparar a captura. A entrada CCInB não está disponível e as outras duas entradas estão conectadas à terra, GND, à VCC. O leitor deve estar estranhando o porquê de ligar entradas à GND e VCC. Isto será explicado no próximo parágrafo.

A saída do MUX1 é entregue a um circuito, denominado de Modo de Captura, que verifica se o flanco especificado aconteceu. São 4 modos de captura, controlados pelos bits CM,

como especificados na Tabela 7.5. Em seu programa, ao alternar entre as entradas 2 (GND) e 3 (VCC) do MUX1, o programador pode provocar um flanco (de subida ou descida) e assim disparar uma captura por *software*.

Continuando com a Figura 7.23, a entrada 0 do MUX2 permite que o programador use diretamente o sinal gerado pela Unidade de Modo de Captura, já a entrada 1 do MUX2, faz a sincronização deste sinal com o relógio do contador TAxR. Como a detecção dos flancos é assíncrona, há o risco de metaestabilidade. Por isso, o manual recomenda, sempre que possível, usar a sincronização (selecionar a entrada 1 do MUX2).

A saída do MUX2 provoca a escrita do valor atual do contador TAxR no registrador de comparação TAxCCRn. Esta mesma saída, através da entrada 1 do MUX3 ativa a *flag* CCIFG. O MUX3, como se nota na figura é controlado pelo bit CAP. Quando CAP = 1, a ativação de CCIFG é feita pela captura e, quando CAP = 0, a ativação de CCIFG é feita pela Unidade de Comparação. Assim, quando no Modo Captura (CAP = 1), a *flag* CCIFG alerta o programador que aconteceu uma captura e que ele pode ler o resultado no registrador de comparação TAxCCRn. Se estiver operando por *polling*, o programador é responsável por zerar esta *flag*, para que ele a perceba novamente indo para 1 na próxima captura.

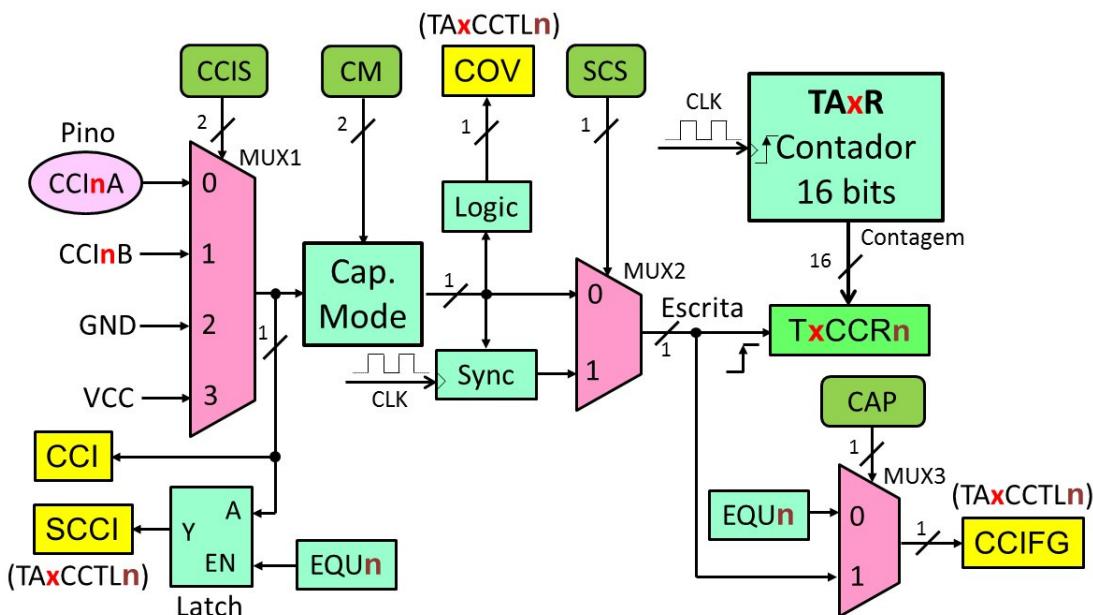


Figura 7.23. Diagrama de blocos para a operação no Modo Captura de Eventos.

Ainda na Figura 7.23, a *flag* COV indica atropelamento na captura, ou seja, aconteceu uma nova captura sendo que o resultado a captura anterior ainda não foi lido. Significa que se perdeu uma captura. O programador é responsável por zerar esta *flag*. Para

terminar, o leitor deve notar o *bit* CCI que permite ao programador, a qualquer momento ler o estado da entrada selecionada para gerar o evento de captura. Já o *bit* SCCI faz a sincronização deste sinal com a saída do comparador. **Para que isso?**

*Tabela 7.5. Modos de Captura disponíveis*

CM	Modo de Captura
0	Sem captura
1	Flanco de subida (↑)
2	Flanco de descida (↓)
3	Ambos os flancos (↔)

*Tabela 7.6. Distribuição das entradas de captura dos timers A nas Portas GPIO para a arquitetura F5529 (em cinza estão os pinos não disponíveis no Launch Pad)*

<b>Timer A0</b>		<b>Timer A1</b>		<b>Timer A2</b>	
TA0.0	P1.1	TA1.0	P1.7	TA2.0	P2.3
TA0.1	P1.2	TA1.1	P2.0	TA2.1	P2.4
TA0.2	P1.3	TA1.2	P2.1	TA2.2	P2.5
TA0.3	P1.4				
TA0.4	P1.5				

### 7.3.5. Interrupções com o Timer A

Cada instância do *Timer A*, tem dedicada a ela duas posições da tabela de vetores de interrupção. Uma posição (vetor) está dedicada exclusivamente ao comparador 0 e a outra (vetor), aos demais comparadores e ao TAIFG. A Figura 7.24 apresenta uma ideia mais clara deste tópico, onde se podem ver as *flags* que provocam interrupções (CCIFG e TAIFG) e as respectivas habilitações (CCIE e TAIE). É claro que se a habilitação está em 0, a interrupção correspondente não acontece. Não se pode esquecer da habilitação geral feita com o *bit* GIE do registrador de estado.

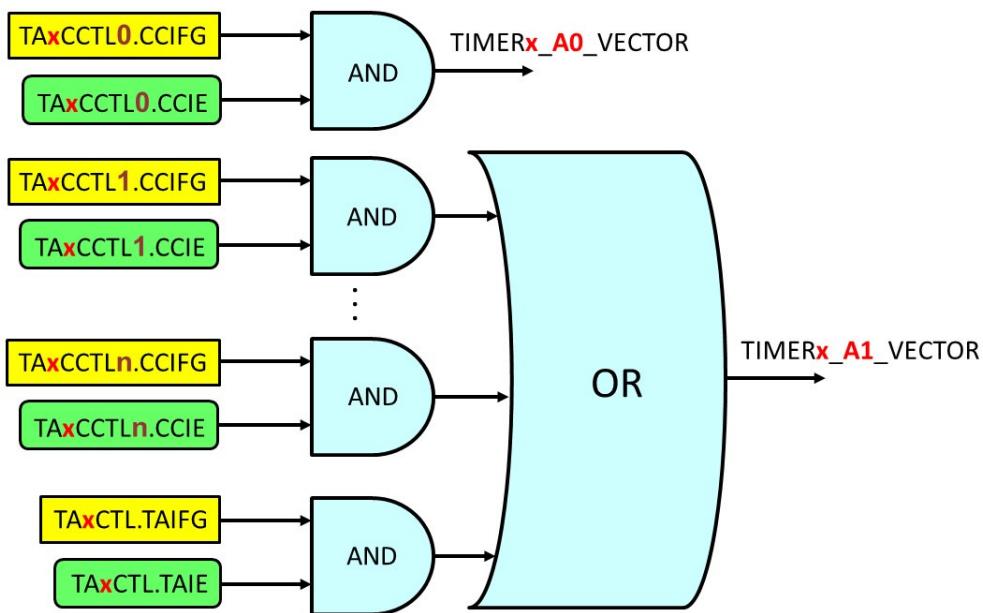


Figura 7.24. Esquema de distribuição dos dois vetores de interrupção para cada instância do Timer A.

A Tabela 7.7 apresenta a lista completa de todas as interrupções dedicadas às diversas instâncias do *Timer A*. Nesta tabela o vetor de maior número tem maior prioridade. O CCS traz um *label* (uma constante) para cada um desses vetores. É importante dar atenção à essa denominação, já que os nomes não foram felizes e podem induzir a erros. Muitas vezes, é mais seguro usar o número do vetor.

Relembrando o comportamento das *flags*, que também podem ser alteradas por software:

- Em modo comparação, TAxCCTLn.CCIFG = 1 quando TAxCRRn = TAxR;
- Em modo captura, TAxCCTLn.CCIFG = 1 quando ocorre a captura programada e
- Em qualquer modo, TAIFG = 1 sempre que TAxR voltar a zero.

Tabela 7.7. Distribuição das posições (entradas) do vetor de interrupções para as diversas instâncias do Timer A

Fonte	Flags	Vetor	Label para o Vetor
Timer A0	TA0CCTL0.CCIFG	53	<i>TIMER0_A0_VECTOR</i>
	TA0CCTL1.CCIFG		
	...		
	TA0CCTL4.CCIFG	52	<i>TIMER0_A1_VECTOR</i>
	TA0CTL.TAIFG		
Timer A1	TA1CCTL0.CCIFG	49	<i>TIMER1_A0_VECTOR</i>
	TA1CCTL1.CCIFG	48	<i>TIMER1_A1_VECTOR</i>

	...		
<i>Timer A2</i>	TA1CCTL2.CCIFG TA1CTL.TAIFG	44	<i>TIMER2_A0_VECTOR</i>
	TA2CCTL0.CCIFG TA2CCTL1.CCIFG ... TA2CCTLn.CCIFG TA2CTL.TAIFG	43	<i>TIMER2_A1_VECTOR</i>

Como vimos, existe um vetor exclusivo para a interrupção do comparador 0 (TAxCCTL0.CCIFG) e sua manipulação é simples, como mostrado na listagem abaixo, que considera o uso do *timer* TA0. Neste caso, quando processador desvia para a rotina de interrupção, a *flag* CCIFG é zerada automaticamente. Se o leitor tiver dúvidas sobre interrupções, é recomendado a leitura do Capítulo 6.

```
// Tratar TA0CCTL0.CCIFG
#pragma vector = 53 // #pragma vector = TIMER0_A0_VECTOR
__interrupt void isr_ta0_ccifg0(void) {
    ;
    Rotina para tratar a interrupção;
    ;
}
```

Por outro lado, existe um único vetor para as demais *flags* de interrupções (CCIFGs e TAIFG), assim, a rotina que trata a interrupção precisa verificar, dentre os possíveis candidatos, quem provocou a interrupção. O leitor pode imaginar que esta verificação vá tomar tempo, pois são vários candidatos. Para facilitar este trabalho o MSP traz o registrador TAxIV. Ele retorna um número (valor) correspondente à interrupção de maior prioridade dentre as *flags* que foram ativadas, como apresentado na Tabela 7.8. Quanto menor o número, maior a prioridade. É de se notar que os valores são números pares. No caso de *assembly*, eles são somados diretamente ao contador de programa (PC) que possui 16 bits. A cada leitura de TAxIV, a *flag* de maior prioridade é automaticamente zerada. Ao retornar da rotina de interrupção, caso haja algum pedido pendente, uma nova interrupção é gerada e assim prossegue até que não haja mais pedidos pendentes.

*Tabela 7.8. Possíveis valores do registrador TAxIV*

Prioridade	Valor	Descrição
-	0	Nenhuma interrupção pendente

Maior	2	TAxCCTL1.CCIFG
	4	TAxCCTL2.CCIFG
	6	TAxCCTL3.CCIFG
	8	TAxCCTL4.CCIFG
	...	...
Menor	-	TAxCTL.TAIFG

A listagem abaixo apresenta uma sugestão para a rotina de interrupção para tratar os pedidos de interrupção de TA0. Relembrando, a função `__even_in_range(TA0IV, 0xA)` garante que o valor lido do registrador TA0IV seja par e esteja dentro da faixa de 0x0 até 0xA. De acordo com o valor retornado, uma determinada função é executada.

```
// Tratar TA0CCTL1.CCIFG, TA0CCTL2.CCIFG, ..., TA0CTL.TAIFG

#pragma vector = 53 // #pragma vector = TIMER0_A1_VECTOR
__interrupt void isr_ta0(void) {
    int n;
    n = __even_in_range(TA0IV, 0xA);
    switch(n) {
        case 0x0: break;
        case 0x0: ta0_ccifg1();      break;
        case 0x2: ta0_ccifg2();      break;
        case 0x4: ta0_ccifg3();      break;
        case 0x6: ta0_ccifg4();      break;
        case 0xA: ta0_taifg();      break;
    }
}

// Tratar TA0CCTL1.CCIFG
void ta0_ccifg1 (void){ ... } // Tratar TA0CCTL1.CCIFG
void ta0_ccifg2(void){ ... } // Tratar TA0CCTL2.CCIFG
void ta0_ccifg3(void){ ... } // Tratar TA0CCTL3.CCIFG
void ta0_ccifg4(void){ ... } // Tratar TA0CCTL4.CCIFG
void ta0_taifg (void){ ... } // Tratar TA0CTL.TAIFG
```

### 7.3.6. Registradores do *Timer A*

A seguir, na Tabela 7.9, estão listados os registradores para o controle e operação do *Timer A*. Logo a seguir se faz a descrição de cada um deles.

Tabela 7.9. Registradores do Timer A

<b>16 bits</b>	<b>Acesso</b>	<b>Reset</b>
TAxCTL	R/W	0
TAxR	R/W	0
TAxCCTLn	R/W	0
TAxCCRn	R/W	0
TAxIV	R/W	0
TAxEX0	R/W	0

### 7.3.6.1. TAxCTL – Registrador de Controle do Timer Ax (*Timer Ax Control Register*)

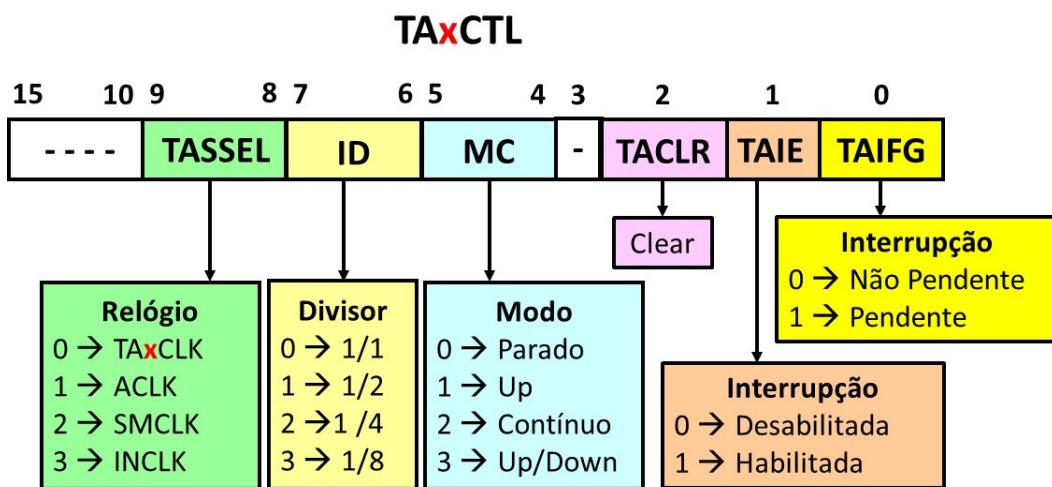


Figura 7.25. Descrição dos bits do registrador TAxCTL (Reset faz TAxCTL=0).

#### (R/W) Bits 15:10: Reservados

#### (R/W) Bits 9:8: TASSEL – Selecionar relógio para TAx (*Timer Ax clock source select*)

O usuário seleciona o relógio que será usado na contagem realizada pelo TAxR.

- TASSEL = 0 → Usar relógio fornecido no pino TAxCLK (ver Figura 7.16), TA0CLK = P1.0, TA1CLK = P1.6 e TA2CLK = P2.2
- TASSEL = 1 → ACLK é selecionado
- TASSEL = 2 → SMCLK é selecionado
- TASSEL = 3 → Não disponível no F5529

**(R/W) Bits 7:6: ID – Divisor do relógio para TAx****(Input divider)**

O usuário indica um divisor para o relógio selecionado (usado junto com o registrador TAxINDEX), que oferece mais divisores.

- ID = 0 → CLK/1
- ID = 1 → CLK/2
- ID = 2 → CLK/3
- ID = 3 → CLK/4

**(R/W) Bits 5:4: MC – Controle do modo de TAx****(Mode control)**

O usuário seleciona um dos 4 modos de contagem disponíveis.

- MC = 0 → Contador parado (economiza energia)
- MC = 1 → Modo ascendente (Up)
- MC = 2 → Modo Contínuo (*Continuous*)
- MC = 3 → Modo Ascendente/Descendente (*Up/down*)

**(R/W) Bit 3: Reservado****(R/W) Bit 2: TACLR – Zeragem do contador TAxR****(Timer A clear)**

Ao ser ativado, este *bit* provoca a zeragem do contador TAxR e do estado interno dos divisores do relógio. Este *bit* volta a zero automaticamente, ou seja, ele é sempre lido como zero.

**(R/W) Bit 1: TAIE – Habilitação da interrupção por ultrapassagem (TAIFG)****(Timer A interrupt enable)**

Quando este *bit* está em 1, a *flag* TAIFG provoca interrupção quando ativada.

**(R/W) Bit 0: TAIFG – Flag de ultrapassagem****(Timer A interrupt flag)**

Esta *flag* vai para 1 de acordo com o modo de contagem selecionado. Ela pode ser consultada (*polling*) pelo programa ou provocar interrupção se o *bit* TAIE estiver em 1. Esta *flag* pode ser apagada por *software*.

**7.3.6.2. TAxR – Contador do Timer Ax****(Timer Ax Register)**



Figura 7.26. Contador de 16 Bits do Timer A (Reset faz  $TAXR = 0$ ).

**(R/W) Bits 15:0: TAXR – Contador de TAX**

**(Timer A register)**

Este é um registrador de 16 bits, responsável por fazer as contagens. Ele pode ser lido ou escrito a qualquer momento.

**7.3.63. TAxCCTLn – Registrador de Controle da instância n da unidade de captura e comparação**

**(Timer Ax Capture/Compare Control n Register)**

**TAxCCTLn (bits 15:8)**

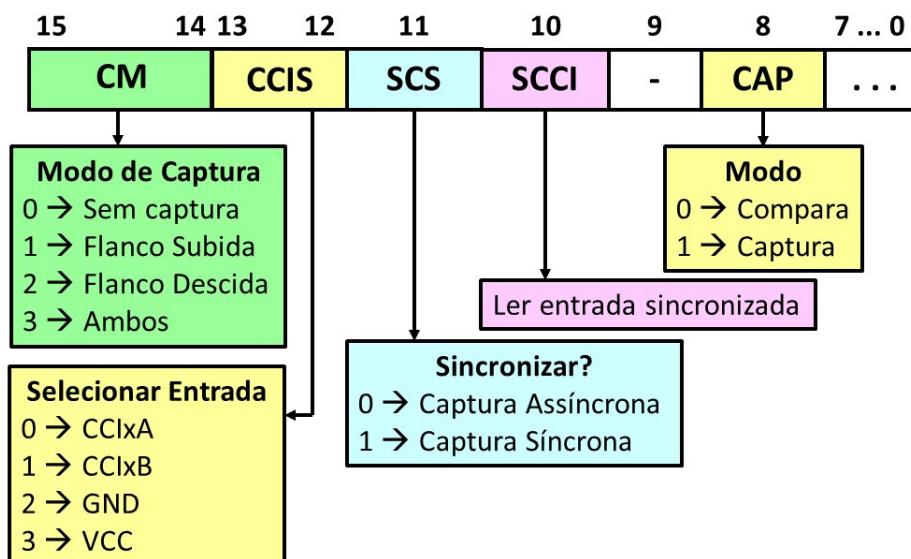


Figura 7.27.a. Descrição dos bits 15:8 do registrador de controle da instância n de captura e comparação do Timer A (Reset faz  $TAxCCTLn= 0$ ).

**(R/W) Bits 15:14: CM – Modo de captura**

**(Capture mode)**

O usuário indica que tipo de flanco caracteriza o evento que ele deseja capturar.

- CM = 0 → Nenhuma captura;
- CM = 1 → (↑) Captura no flanco de subida
- CM = 2 → (↓) Captura no flanco de descida
- CM = 3 → (↔) Captura em ambos os flancos

**(R/W) Bits 13:12: CCIS – Seleção da entrada de captura  
(Capture/Compare input select)**

O usuário seleciona por qual entrada ele vai apresentar o evento a ser capturado. Na verdade, apenas a entrada CCIAx está disponível externamente. Na versão F5529, a entrada CCIBx não está disponível. Veja Tabela 7.6 para identificar os pinos que disponibilizam as entradas CCIAx para cada instância do *Timer A*. A alternância entre as opções 2 (GND) e 3 (VCC) permite que, por *software*, se provoque um flanco de subida ou de descida.

- CM = 0 → CCIAx
- CM = 1 → CCIBx
- CM = 2 → GND (terra)
- CM = 3 → VCC (alimentação)

**(R/W) Bit 11: SCS – Sincronizar entrada de captura  
(Synchronize Capture source)**

Quando este *bit* está em 1, o sinal a ser capturado é sincronizado com o relógio de TAx. Com este *bit* em 0, o sinal a ser capturado é assíncrono com o contador TAxR, o que pode resultar em metaestabilidade. É recomendado sempre habilitar esta sincronização.

**(R/W) Bit 10: SCCI – Ler entrada de captura sincronizada  
(Synchronized capture/compare input)**

Por este *bit* se pode ler a entrada de captura, amostrada no instante em que o sinal interno EQUn é ativado. **Para que serve isso? Quando opera no modo Up, permite saber se aconteceu captura?**

**(R/W) Bit 9: Reservado**

**(R/W) Bit 8: CAP – Seleção do modo captura  
(Capture mode)**

Por este *bit* o usuário escolhe se quer usar o Modo de Comparação (CAP = 0) ou o Modo de Captura (CAP = 1).

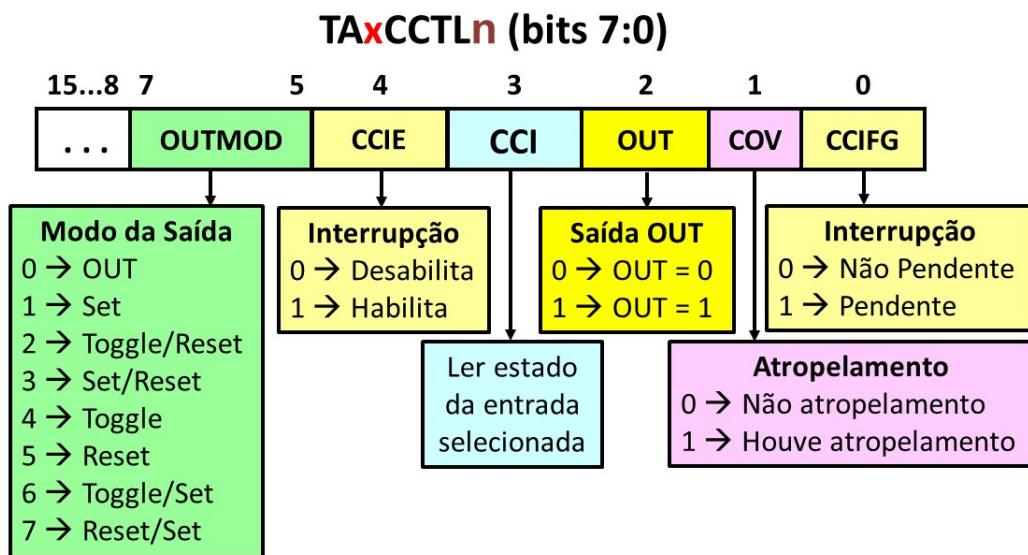


Figura 7.27.b. Descrição dos Bits 7:0 do registrador de controle da instância n de captura e comparação do Timer A (Reset faz TAxCCTL<sub>n</sub>= 0).

**(R/W) Bits 7:5: OUTMOD – Seleção do modo de saída**

**(Output mode)**

O usuário seleciona como deve operar a unidade geradora de saída. Ver as Figura 7.19, 7.20 e 7.21. Com OUTMOD = 0, a saída acompanha o estado do bit 2 deste registrador, denominado OUT.

- OUTMOD = 0 → Acompanha bit OUT
- OUTMOD = 1 → Set
- OUTMOD = 2 → Toggle/Reset
- OUTMOD = 3 → Set/Reset
- OUTMOD = 4 → Toggle
- OUTMOD = 5 → Reset
- OUTMOD = 6 → Toggle/Set
- OUTMOD = 7 → Reset/Set

**(R/W) Bit 4: CCIE – Habilitação da interrupção por captura/comparação (CCIFG)**  
**(Capture/Compare interrupt enable)**

Quando este bit está em 1, a flag CCIFG provoca interrupção quando ativada.

**(R/W) Bit 3: CCI – Entrada de captura**

**(Capture/compare input)**

Por este bit se pode ler o estado instantâneo da entrada de captura.

**(R/W) Bit 2: OUT – Saída**

**(Output)**

Com OUTMOD = 0, a saída acompanha o estado deste bit.

**(R/W) Bit 1: COV – Atropelamento na captura**

**(Capture overflow)**

Esta *flag* vai para 1 quando um novo evento de captura acontece antes do resultado da captura anterior ser lido. Esta *flag* precisa ser zerada por *software*.

#### (R/W) Bit 0: CCIFG – *Flag de captura/comparação*

##### (*Capture/compare interrupt flag*)

Quando no Modo Comparação, esta *flag* vai para 1 para indicar que o valor do contador TAxR ficou igual ao do registrador de comparação TAxCCRn. Quando no Modo Captura, esta *flag* vai para 1 para indicar que o evento de captura selecionado ocorreu e que o resultado está disponível no registrador TAxCCRn. Esta *flag* pode provocar interrupção quando o bit CCIE está em 1. Ela pode ser apagada por *software*.

#### 7.3.6.4. TAxCCRn – Reg. da Unidade n de Captura/Comparação do Timer Ax (*Timer Ax Capture/Compare n Register*)

##### TAxCCRn



Figura 7.28. Registrador da instância n da Unidade de Captura/Comparação do timer TAx.  
(Reset faz TAxCCRn= 0).

#### (R/W) Bits 15:0: TAxCCRn – Registrador da Unidade n de Captura/Comparação (*Capture/Compare n Register*)

Se no modo comparação (CAP = 0), este registrador armazena o valor a ser comparado com o contador TAxR. Se no modo captura (CAP = 1), conteúdo do contador TAxR é copiado para este registrador, quando ocorre o evento de captura especificado.

#### 7.3.6.5. TAxIV – Registrador de Vetor de Interrupção do Timer Ax (*Timer Ax Interrupt Vector Register*)

##### TAxIV



Figura 7.29. Registrador de Vetor de Interrupção do timer TAx. (Reset faz TAxIV= 0).

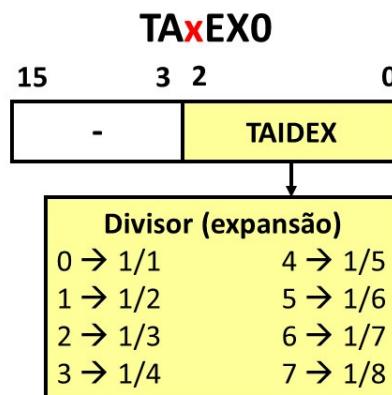
**(R) Bits 15:0: TAxCCRn – Valor do Vetor de Interrupção  
(Interrupt Value Register)**

A leitura deste registrador retorna um número que indica, dentre as interrupções pendentes, a de maior prioridade e, ao mesmo tempo, apaga este pedido pendente (o de maior prioridade). A tabela abaixo indica as opções para as instâncias do *Timer A* disponíveis na versão F5529.

*Tabela 7.10. Possíveis valores do Registrador TAxIV, de acordo com as instâncias do Timer A disponíveis na versão F5529.*

Valor	Timer A0	Timer A1	Timer A2
0x0	Nada pendente	Nada pendente	Nada pendente
0x2	TA0CCR1.CCIFG	TA1CCR1.CCIFG	TA2CCR2.CCIFG
0x4	TA0CCR2.CCIFG	TA1CCR2.CCIFG	TA2CCR3.CCIFG
0x6	TA0CCR3.CCIFG	-	-
0x8	TA0CCR4.CCIFG	-	-
0xA	-	-	-
0xC	-	-	-
0xE	TA0CTL.TAIFG	TA1CTL.TAIFG	TA2CTL.TAIFG

**7.3.6.6. TAxE0 – Registrador 0 para Expansão de *Timer Ax*  
(*Timer Ax Expansion 0 Register*)**



*Figura 7.30. Registrador 0 para expansão do Timer TAx. (Reset faz TAxE0 = 0).*

**(R) Bits 15:3: Reservados**

**(R/W) Bits 2:0: TAIDEX – Expansão do divisor do relógio****(Input divider expansion)**

Esses 3 *bits* oferecem mais opções para a divisão do relógio que vai acionar o contador TAxR. O valor máximo de divisão que se pode obter é igual a 64 ( $ID = 3 \rightarrow 1/8$  e  $TAIDEX = 7 \rightarrow 1/8$ ). Vide Figura 7.16.

## 7.4. Estudo do *Timer B*

O MSP430F5529 oferece apenas a instância 0 (zero) do *Timer B*, com 7 instâncias das Unidades de Captura/Comparação. O *Timer B* tem muita semelhança com o *Timer A*, razão pela qual se recomenda o estudo do tópico anterior, que aborda com detalhes o *Timer A*. Aqui neste tópico serão apresentadas apenas as diferenças entre estes dois *timers*.

O *Timer B* apresenta as seguintes diferenças com relação ao *Timer A*:

- O tamanho do contador (TB0R) é configurável para 8, 10, 12 ou 16 *bits*;
- Os registradores TB0CCRn têm dupla buferização e podem ser agrupados;
- Todas as saídas do *Timer B*0 podem entrar no estado de alta-impedância e
- O *bit* SCCI não está disponível.

Já vimos que existem registradores para controlar a operação do contador e para controlar o funcionamento das unidades de captura e comparação. Como na versão F5529 está disponível apenas a instância zero, os nomes dos registradores sempre serão iniciados com TB0 (e não TBx). A tabela abaixo resume os registradores deste *timer*. O controle do contador é feito pelo registrador TB0CTL. Os diversos registradores de controle das unidades Captura/Compara são denominados TB0CCTLn, onde n = 0, 1, ...6.

*Tabela 7.11. Resumo dos registradores das instâncias do Timer B e de seus registradores de Captura e Compara*

Finalidade	Registrador	Controle	Flag IFG
Contador	TB0R	TB0CTL	TAIFG
Captura/Compara	TB0CCRn	TB0CCTLn	CCIFG

### 7.4.1. Excursão da Contagem do *Timer B*0

O contador TB0R pode ser configurado para operar com 8, 10, 12 ou 16 *bits*. Estas opções são selecionadas com os *bits* CNTL do registrador TB0CTL. Os valores escritos no registrador TB0R, quando este opera com menos de 16 *bits*, são alinhados pela direita

e a porção da esquerda é preenchida com zeros, como seria esperado. Abaixo estão listados os valores máximos de contagem, para cada excursão disponível.

- 8 bits → máximo = 0x00FF;
- 10 bits → máximo = 0x03FF;
- 12 bits → máximo = 0x0FFF;
- 16 bits → máximo = 0xFFFF;

### 7.4.2. Dupla Buferização do *Timer B0*

A necessidade da dupla buferização se explica pela possibilidade de que, durante a geração de PWM, uma alteração no registrador de comparação gere pulsos de largura não desejada. Para explicar melhor, vamos considerar a situação apresentada na Figura 7.31. Nesta figura o contador está operando no Modo Ascendente (*Up*) e o registrador TB0CCRn é usado para controlar o ciclo de carga do PWM. Podemos ver que o valor de TB0CCRn era “q” e foi alterado para “p”, sendo  $p < q$ . Porém, essa troca de valor pode acontecer num instante ruim, como ilustrado nesta figura. Note que o contador TB0R estava entre “p” e “q” quando a troca aconteceu. Como TB0CCRn foi alterado para um valor abaixo do valor atual de TB0R, este vai contar até o limite (valor de TB0CCR0) e durante este período de contagem nunca teremos a coincidência entre o contador e o comparador ( $TB0R = TB0CCRn$ ). Isto significa que a saída permanece em 1, gerando um pulso largo, diferente do que se desejava. A dupla buferização evita este tipo de problema.

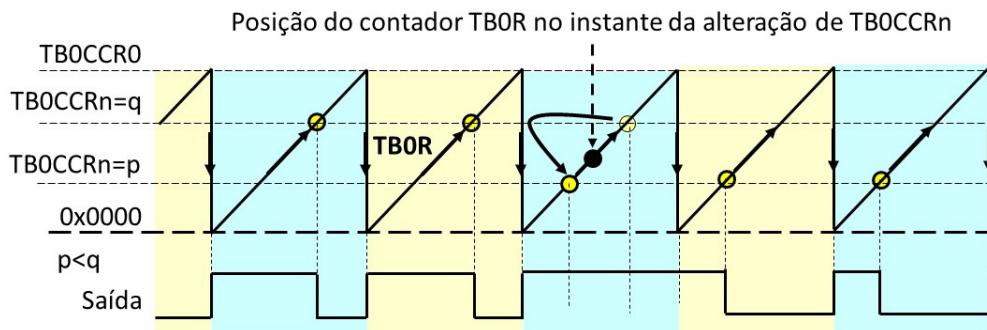


Figura 7.31. Exemplo de uma saída inesperada que pode acontecer quando se altera do registrador de comparação (TB0CCRn), se não houvesse o recurso de dupla buferização.

Na Figura 7.32 é possível de ver que o programador não tem acesso direto ao registrador TB0CLn, que é usado na comparação. Para mudar o valor deste registrador, o programador deve escrever no registrador TB0CCRn e, de acordo com o que foi programado nos bits CLLD, o valor de TB0CCRn é transferido para TB0CLn num instante de tempo muito bem determinado. Com o uso deste recurso, o problema indicado na

Figura 7.31 não aconteceria. A Tabela 7.12 apresenta as opções para a seleção do instante desta transferência.

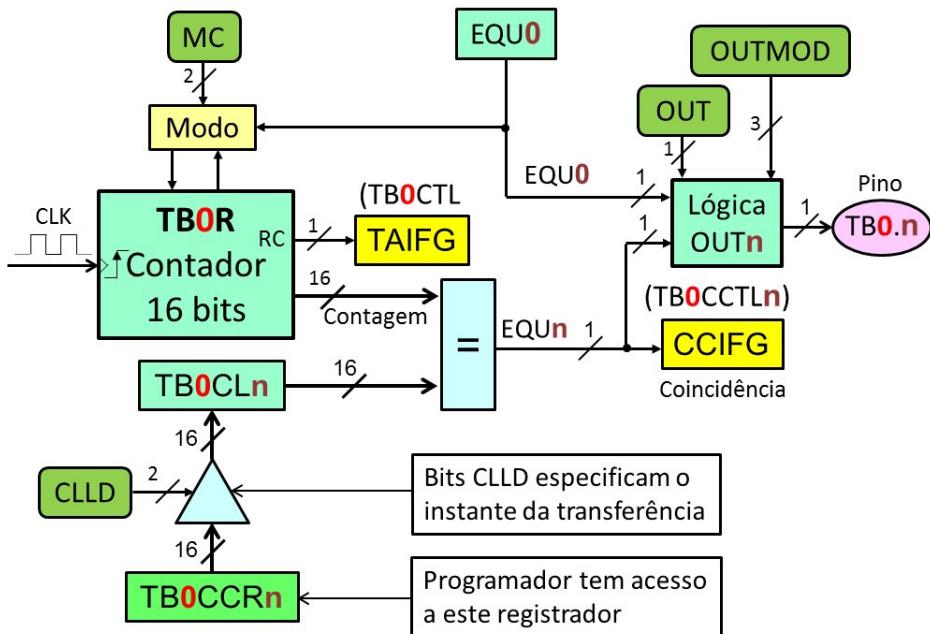


Figura 7.32. Diagrama de blocos para a Unidade Geradora de Saídas (Lógica OUTn) do comparador n, indicando a lógica de dupla buferização.

Tabela 7.12. Definição do instante de atualização do registrador TB0CTLn.

CLLD	Transferência	Descrição
0		Imediata
1		TB0R = 0
2	TB0CCRn → TB0CLn	(MC = 1 ou 2) TB0R = 0 (MC = 3) TB0R = 0 ou TB0R = TB0CL0 antigo
3		TB0R = TB0CL0 antigo

#### 7.4.3. Agrupamento de Registradores de Comparaçāo do Timer B0

Algumas aplicações podem exigir que o usuário altere, simultaneamente, dois registradores de comparação. Por exemplo, imaginemos o caso de se alterar o ciclo de carga de duas saídas PWM que precisam estar sincronizadas. Mesmo usando a dupla buferização, pode ser que essas duas alterações fiquem separadas por um período de contagem. Com o que foi visto até agora, é impossível garantir que dois registradores

sempre sejam alterados ao mesmo tempo, já que se faz necessário o uso de duas instruções de escrita.

Para solucionar esta necessidade, o *hardware* do *Timer B* faz uso da dupla buferização, mas agora o instante da transferência dos valores é dado pela escrita num determinado registrador. A isso se denomina agrupamento. Por exemplo, os registradores de comparação TB0CCR1 e TB0CCR2 podem ser agrupados, assim o usuário escreve em TB0CCR2 e depois em TB0CCR1. Quando acontecer a segunda escrita, os dois registradores (TB0CL1 e TB0CL2) são atualizados, simultaneamente.

Quando se trabalha com agrupamento de registradores, a escrita no registrador de menor número define o momento da atualização de todo o agrupamento. A Tabela 7.13 apresenta as opções de agrupamento. Há uma particularidade. Note que na última opção, todos os registradores de comparação estão agrupados, incluindo TB0CTL0, e que o controle é feito pela escrita em TB0CCR1.

*Tabela 7.13. Opções de agrupamento de registradores*

TBCLGRP0	Agrupamento	Controle
0	Nenhum	Individual
1	TB0CL1 + TB0CL2 TB0CL3 + TB0CL4 TB0CL5 + TB0CL6	TB0CCR1 TB0CCR3 TB0CCR5
2	TB0CL1 + TB0CL2 + TB0CL3 TB0CL4 + TB0CL5 + TB0CL6	TB0CCR1 TB0CCR4
3	TB0CL0 + TB0CL1 + ... + TB0CL6	TB0CCR1

#### 7.4.4. Registradores do *Timer B*

Na Tabela 7.14, logo a seguir, estão listados os registradores para o controle e operação do *Timer B*. Como já foi afirmado, a versão F5529 oferece apenas a instância zero, todos esses registradores iniciam com as letras TB0 (e não TBx).

*Tabela 7.14. Registradores do Timer B0*

16 bits	Acesso	Reset
TB0CTL	R/W	0
TB0R	R/W	0
TB0CCTLn	R/W	0
TB0CCRN	R/W	0

TB0IV	R/W	0
TB0EX0	R/W	0

A seguir são estudados os *bits* desses diversos registradores. Como é grande a semelhança com o *Timer A*, apenas pontos divergentes serão detalhados.

#### 7.4.4.1. TB0CTL – Registrador de Controle do *Timer B0* (*Timer B0 Control Register*)

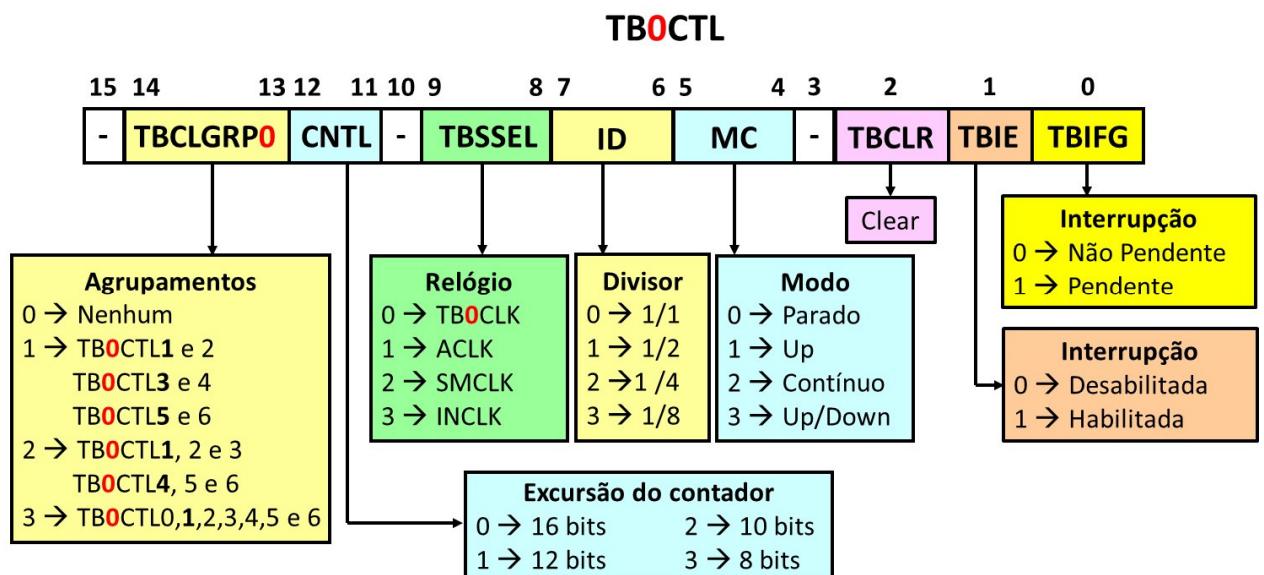


Figura 7.33. Descrição dos Bits do registrador TBxCTL (Reset faz TBxCTL=0).

##### (R/W) Bits 14:13: TBCLGRP0 – Controle de agrupamento (TB0CLn Group)

O usuário seleciona o agrupamento que deseja para os registradores de captura/contagem. Nos modos de agrupamento 1 e 2, a escrita no registrador de menor número dispara a escrita em todo o grupo. Note que na opção 3 estão agrupados todos os registradores de captura/contagem.

- TBCLGRP0 = 0 → Nenhum agrupamento;
- TBCLGRP0 = 1 → (TB0CCR1) TB0CTL1 + TB0CTL2, (TB0CCR3) TB0CTL3 + TB0CTL4, (TB0CCR5) TB0CTL5 + TB0CTL6. A escrita no registrador entre parêntesis dispara a escrita em todo o grupo.
- TBCLGRP0 = 2 → (TB0CCR1) TB0CTL1 + TB0CTL2 + TB0CTL3, (TB0CCR4) TB0CTL4 + TB0CTL5 + TB0CTL6. A escrita no registrador entre parêntesis dispara a escrita em todo o grupo.

- $TBCLGRP0 = 3 \rightarrow TB0CTL0 + (TB0CCR1) TB0CTL1 + TB0CTL2 + TB0CTL3 + TB0CTL4 + TB0CTL5 + TB0CTL6$ . A escrita no registrador entre parêntesis dispara a escrita em todo o grupo.

**(R/W) Bits 12:11: CNTL – Controle da excursão da contagem**

**(Counter lenght)**

O usuário seleciona o tamanho (número de *bits*) do contador TB0R.

- CNTL = 0 → 16 *bits*, máximo com TB0R = 0xFFFF
- CNTL = 1 → 12 *bits*, máximo com TB0R = 0xFFFF
- CNTL = 2 → 10 *bits*, máximo com TB0R = 0x3FF
- CNTL = 3 → 8 *bits*, máximo com TB0R = 0xFF

**7.4.4.2. TB0R – Contador do *Timer B*, instância 0**

**(*Timer B0 Register*)**



Figura 7.34. Contador de 16 Bits do Timer B, instância 0 (Reset faz TB0R= 0).

**7.4.4.3. TB0CCTL<sub>n</sub> – Registrador de Controle da instância n da unidade de captura e comparação do Timer B0**

**(*Timer B0 Capture/Compare Control n Register*)**

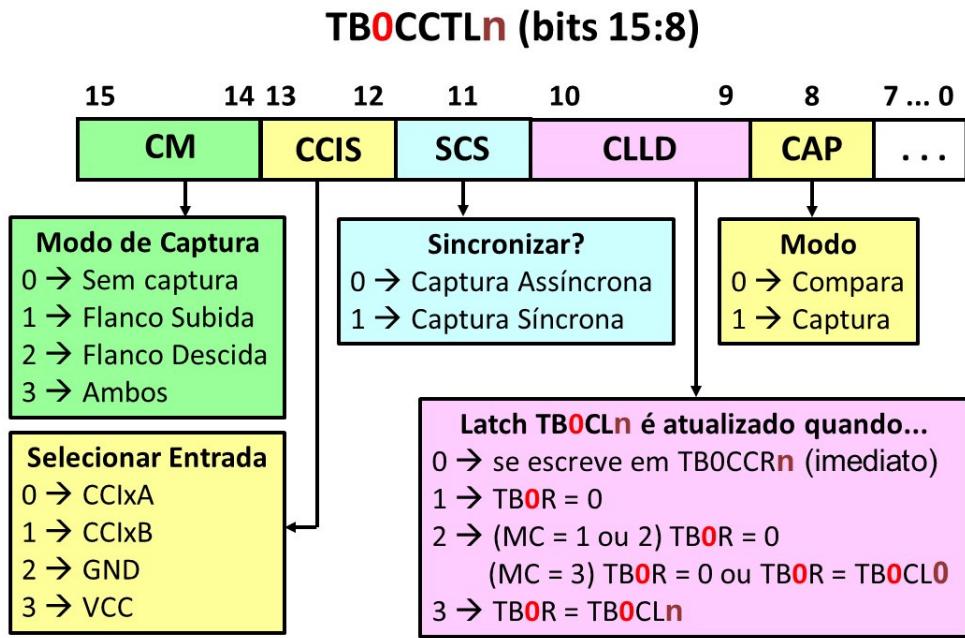


Figura 7.35.a. Descrição dos Bits 15:8 do registrador de controle da instância n de captura e comparação do Timer B0 (Reset faz TBxCCTL<sub>n</sub> = 0).

#### (R/W) Bits 10:9: CCLD – Atualização do latch de comparação

##### (Compare latch load)

Com esses Bits o usuário seleciona o instante em que o valor do registrador TB0CCR<sub>n</sub>, que foi atualizado pelo programador, é copiado para o latch TB0CL<sub>n</sub>.

- CLLD = 0 → a escrita em TB0CCR<sub>n</sub> atualiza imediatamente o latch TB0CL<sub>n</sub>, isto significa que a dupla bufferização não é usada.
- CLLD = 1 → o latch TB0CL<sub>n</sub> é atualizado quando o contador (TB0R) chega a zero.
- CLLD = 2 → se o contador estiver no Modo Ascendente (MC = 1) ou no Modo Contínuo (MC = 2), o latch TB0CL<sub>n</sub> é atualizado quando o contador (TB0R) chega a zero, porém se o Modo Ascendente/Descendente (MC = 3) estiver selecionado, a atualização ocorrerá quando o contador (TB0R) chegar a zero ou quando ele for igual a TB0CL0.
- CLLD = 3 → o latch TB0CL<sub>n</sub> é atualizado quando o contador (TB0R) ficar igual a TB0CL<sub>n</sub> (valor anterior).

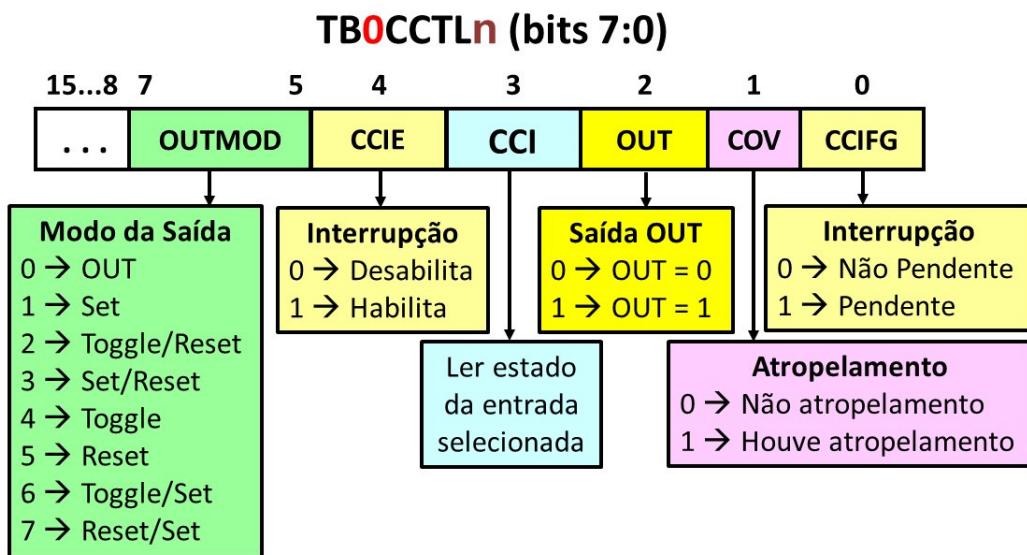


Figura 7.35.b. Descrição dos Bits 7:0 do registrador de controle da instância n de captura e comparação do Timer B0 (Reset faz  $TBxCCTL_n=0$ ).

#### 7.4.4.4. TB0CCR<sub>n</sub> – Reg. da Unidade n de Captura/Comparaçāo do Timer B0 (Timer B0 Capture/Compare n Register)

#### TB0CCR<sub>n</sub>



Figura 7.36. Registrador da instância n da Unidade de Captura/Comparaçāo do timer TB0.  
(Reset faz  $TB0CCR_n=0$ ).

#### 7.4.4.5. TB0IV – Registrador de Vetor de Interrupçāo do Timer B0 (Timer B0 Interrupt Vector Register)

#### TB0IV



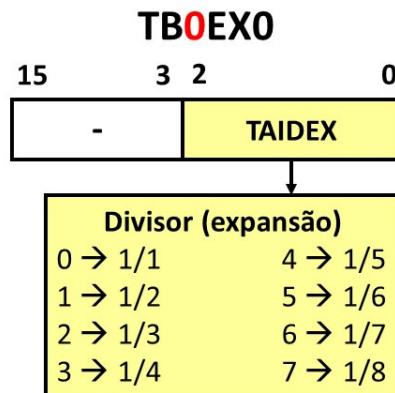
Figura 7.37. Registrador de Vetor de Interrupçāo do timer TB0. (Reset faz  $TB0IV=0$ ).

**(R) Bits 15:0: TB0IV – Valor do Vetor de Interrupção****(Interrupt Value Register)**

A leitura deste registrador retorna um número que indica a interrupção pendente de maior prioridade, ao mesmo tempo em que apaga este pedido pendente. A tabela abaixo indica as opções para a instância 0 do *Timer B*, a única disponível na versão F5529.

*Tabela 7.15. Possíveis valores do Registrador TB0IV*

Valor	Timer B0
0x0	Nada pendente
0x2	TB0CCR1.CCIFG
0x4	TB0CCR2.CCIFG
0x6	TB0CCR3.CCIFG
0x8	TB0CCR4.CCIFG
0xA	TB0CCR5.CCIFG
0xD	TB0CCR6.CCIFG
0xE	TA0CTL.TAIFG

**7.4.4.6. TB0EX0 – Registrador 0 para Expansão de Timer B0**  
**(Timer B0 Expansion 0 Register)**


*Figura 7.38. Registrador 0 para expansão do timer TB0. (Reset faz TB0EX0 = 0).*

---

**7.5. Exercícios Resolvidos**


---

A seguir são apresentados diversos exercício resolvidos. O leitor é convidado a executar todos eles usando o Code Composer Studio (CCS) e a placa **MSP430 F5529 LaunchPad**.

De nada adianta simplesmente ler os problemas e as soluções. Seria como tentar aprender a andar de bicicleta apenas observando os ciclistas no parque. É preciso que o leitor tente esboçar sua solução e depois a compare com a solução apresentada. É necessário desenvolver a habilidade de *pensar com os recursos de temporização do MSP*.

Como o *Timer A* e o *Timer B* são semelhantes, em sua grande maioria os exercícios resolvidos fazem uso do *Timer A0*. Ao final são apresentados alguns exercícios que lançam mão das particularidades do *Timer B0*

**ER 7.1.** Usando o ACLK (32.678 Hz), escreva um programa, que constrói um relógio de horas, minutos, segundos e décimos de segundos usando o TA0. De alguma forma, use os *leds* para indicar o funcionamento do relógio.

#### Solução:

Precisamos de encontrar uma forma de fazer uma *flag* qualquer do *timer TA0* ir para nível alto a cada 0,1 s. Foi pedido para usar o ACLK que faz 32.768 contagens em 1 segundo. Isto significa que em 0,1 segundo (um décimo de segundo) ele faz 3.277 contagens, aproximadamente. A forma mais simples de se conseguir que uma *flag* indo para 1 a cada 0,1 s é usar o Modo Ascendente (MC = 1 - *up*) e limitar a contagem com TA0CCR0 em 3.276 (3.277 - 1), pois o zero também é contado, como mostrado na Figura 7.11. Assim, basta monitorar a *flag* TA0CCR0.CCIFG como apresentado na listagem abaixo.

#### Listagem solução do ER 7.1

```
// ER 7.1
// Relógio com horas, minutos, segundos e décimos de segundos
// Usar o ACLK = 32.768 Hz

#include <msp430.h>

#define TRUE      1
#define DEC_SEG  3276      //Contagens de ACLK para um décimo de segundo

void config_leds(void);

int main(void){
```

```

unsigned int dseg,seg,min,hora;
WDTCTL = WDTPW | WDTHOLD;      // stop watchdog timer

config_leds();
TA0CTL = TASSEL_1 |      //Selecionar ACLK
          MC_1;           //Modo Ascendente (UP)
TA0CCR0 = DEC_SEG;
dseg=seg=min=hora=0;
while(TRUE) {
    while( (TA0CCTL0&CCIFG) == 0); //Esperar CCIFG = 1
    TA0CCTL0 &= ~CCIFG;           //Apagar a flag CCIFG
    P4OUT ^= BIT7;               //0,1s --> Inverter led Verde
    if (++dseg == 10){           //10 dec_seg?
        dseg = 0;
        P1OUT ^= BIT0;           //1 seg-->Inverter led Vermelho
        if (++seg == 60){         //60 seg?
            seg = 0;
            if (++min == 60){     //60 min?
                min = 0;
                if (++hora == 24)  //24 horas?
                    hora = 0;
            }
        }
    }
    return 0;
}

void config_leds(void) {
    P1DIR |= BIT0;             //(Verm) P1.0 = saída
    P1OUT &= ~BIT0;            //Verm apagado
    P4DIR |= BIT7;             //(Verde) P1.7 = saída
    P4OUT &= ~BIT7;            //Verde apagado
}

```

O leitor também poderia monitorar a *flag* TA0CTL.TAIFG, para tanto, bastaria alterar, na listagem acima, as duas linhas marcadas em negrito, para:

```

while( (TA0CTL&TAIFG) == 0); //Esperar TAIFG = 1
TA0CTL &= ~TAIFG;           //Apagar a flag TAIFG

```

Como ainda não temos nenhum dispositivo de saída para visualizar o relógio, usou-se o recurso dos *leds*, de acordo com o pedido. O estado do *led* verde é invertido a cada 0,1 s, ou seja, ele pisca na frequência de 5 Hz. Já o estado do *led* vermelho é invertido a cada 1,0 s, ou seja, ele pisca na frequência de 0,5 Hz. Usando um cronômetro e contando as “piscadas” de um dos *leds*, o leitor pode tentar conferir o relógio.

Desafio: O valor de TA0CCR0 deveria ser igual ao calculado abaixo. Entretanto, não é possível carregar valores fracionários nos registradores, assim, o resultado da divisão foi arredondado para 3.276. Com isso o relógio atrasa ou adianta? Qual o erro em 1 hora?

$$TA0CCR0 = \frac{32.768}{10} - 1 = 3.275,8$$

Resposta: O relógio atrasa 2,197 segundos a cada hora.

**ER 7.2.** Usando o SMCLK (1.048.576 Hz), escreva um programa, que constrói um relógio de horas, minutos, segundos e décimos de segundos usando o TA0. De alguma forma, use os *leds* para indicar o funcionamento do relógio (é semelhante ao anterior).

#### Solução:

A solução semelhante à do exercício anterior. Se o SMCLK faz 1.048.576 contagens em 1 segundo. Isto significa que em 0,1 segundo ele faz 104.858 contagens, aproximadamente. Será que fica tudo igual e basta fazer  $TA0CCR0 = 104.857$ ?

É preciso lembrar que os registradores de TA0 são de 16 bits, assim, o maior valor que eles aceitam é 65.535. Neste caso, é preciso diminuir a frequência do relógio que está sendo usado. Podemos, então, dividir o SMCLK por 2, e assim ficamos com 524.288 contagens em 1 segundo. Finalmente, fazemos  $TA0CCR0 = 52.428$ . A listagem apresenta o programa correspondente. Note que, para economizar espaço, a função *config\_leds()* foi omitida e deve ser copiada da listagem anterior.

#### Listagem solução do ER 7.2

```
// ER 7.2
// Relógio com horas, minutos, segundos e décimos de segundos
// Usar o SMCLK = 1.048.576 Hz

#include <msp430.h>

#define TRUE      1
#define DEC_SEG  52428     //Contagens de SMCLK/2 para um décimo de segundo

void config_leds(void);

int main(void) {
    unsigned int dseg,seg,min,hora;
    WDTCTL = WDTPW | WDTHOLD;    // stop watchdog timer

    config_leds();
    TA0CTL = TASSEL_2 |      //Selecionar SMCLK
              ID_1        |      //Divisor por 2
              MC_1;          //Modo Ascendente (UP)
```

```

TA0CCR0 = DEC_SEG;
dseg=seg=min=hora=0;
while(TRUE) {
    while( (TA0CCTL0&CCIFG) == 0); //Esperar CCIFG = 1
    TA0CCTL0 &= ~CCIFG;           //Apagar a flag CCIFG
    P4OUT ^= BIT7;               //0,1 s --> Inverter led Verde
    if (++dseg == 10){          //10 dec_seg?
        dseg = 0;
        P1OUT ^= BIT0;           //1 seg --> Inverter led Vermelho
        if (++seg == 60){        //60 seg?
            seg = 0;
            if (++min == 60){   //60 min?
                min = 0;
                if (++hora == 24) //24 horas?
                    hora = 0;
            }
        }
    }
    return 0;
}
void config_leds(void){...} //Copiar do ER 7.1

```

Desafio: O valor de TA0CCR0 deveria ser igual ao calculado abaixo. Entretanto, não é possível usar valores fracionários, assim, foi arredondado para 52.428. Com isso o relógio atrasa ou adianta? Qual o erro em 1 hora? Ele ficou mais preciso que a solução anterior?

$$TA0CCR0 = \frac{524.288}{10} - 1 = 52.427,8$$

Resposta: O relógio atrasa 0,137 segundos a cada hora, erro 16 vezes menor que o do exercício anterior. Por isso, esta solução é mais precisa.

**ER 7.3.** Usando interrupção e o SMCLK (1.048.576 Hz), escreva um programa, que constrói um relógio de horas, minutos, segundos e décimos de segundos usando o TA0.

### Solução:

Esta solução lembra a do ER 7.2, mas ao invés de ficarmos esperando uma *flag* ir para 1, vamos fazer uso das interrupções. Vimos que temos duas formas de construir este relógio, a primeira usando a *flag* do comparador 0 (TA0CCTL0.CCIFG) e a outra é usando a indicação de quando o contador passa pelo zero (TA0CTL.TAIFG).

Como primeira solução, vamos usar a interrupção do comparador 0, pois a arquitetura MSP430 tem um vetor exclusivo para esta interrupção, vide Figura 7.13. A listagem abaixo apresenta essa solução.

### Listagem solução do ER 7.3.a

```

// ER 7.3a
// Relógio com horas, minutos, segundos e décimos de segundos
// Usar o SMCLK = 1.048.576 Hz e interrupção com TA0CCR0.CCIFG

#include <msp430.h>

#define TRUE      1
#define DEC_SEG 52428 //Contagens de SMCLK/2 para um décimo de segundo

void config_leds(void);

volatile unsigned int dseg,seg,min,hora;

int main(void){
    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer

    config_leds();
    TA0CTL = TASSEL_2 | //Selecionar SMCLK
              ID_1     | //Divisor por 2
              MC_1;      //Modo Ascendente (UP)
    TA0CCR0 = DEC_SEG;
    TA0CCTL0 = CCIE; //Habilitar interrupção CCIFG
    __enable_interrupt(); //Hab. Geral (GIE=1)
    dseg=seg=min=hora=0;
    while(TRUE); //Programa fica parado
    return 0;
}

// Rotina de interrupção
// Acontece a cada 0,1 segundo
//#pragma vector = 53
#pragma vector = TIMER0_A0_VECTOR
_interrupt void isr_ta0_ccifg0(void){
    P4OUT ^= BIT7; //0,1 s --> Inverter led Verde
    if (++dseg == 10){ //10 dec_seg?
        dseg = 0;
        P1OUT ^= BIT0; //1 seg --> Inverter led Vermelho
        if (++seg == 60){ //60 seg?
            seg = 0;
            if (++min == 60){ //60 min?
                min = 0;
                if (++hora == 24) //24 horas?
                    hora = 0;
            }
        }
    }
}

```

```
void config_leds(void) { ... } //Copiar do ER 7.1
```

Na solução apresentada, é importante salientar a habilitação da interrupção do comparador 0 que tem origem na flag CCIFG, foi feita na linha `TA0CCTL0 = CCIE`. A linha `_enable_interrupt()` faz a habilitação geral de interrupção (`GIE = 1`). A função que atende à interrupção recebeu o nome `isr_ta0_ccifg0`. Poder-se-ia usar qualquer nome, entretanto a intenção foi deixar claro que esta função atende à interrupção do comparador 0 (`TA0CCTL0.CCIFG`). Uma consulta à Tabela 7.7 (Tabela de Vetores de Interrupção) vai indicar que a posição 53 foi reservada para esta interrupção.

Nas duas linhas abaixo, recortadas da solução apresentada, indica-se para o compilador que o endereço da função `isr_ta0_ccifg0` deve ser colocada na posição 53 da Tabela de Vetores de Interrupção. O usuário precisa lembrar que o número “53” especifica uma determinada posição nessa tabela.

```
#pragma vector = 53
__interrupt void isr_ta0_ccifg0(void) { ... }
```

Para aumentar a legibilidade dos programas, o arquivo MSP430.h nomeia as posições dessa tabela, assim, `TIMER0_A0_VECTOR` é uma constante cujo valor é 53. Então, as duas linhas abaixo apresentam uma outra forma de se especificar a posição na Tabela de Vetores de Interrupção.

```
#pragma vector = TIMER0_A0_VECTOR
__interrupt void isr_ta0_ccifg0(void) { ... }
```

É indiferente usar qualquer uma das duas opções. Os autores deste texto consideram denominação dos vetores dos *timers* um pouco infeliz, pois é fácil se enganar e considerar que a constante `TIMER0_A1_VECTOR` se refere ao *timer* TA1.

- `TIMER0_A0_VECTOR` → TA0CCTL0.CCIFG e
- `TIMER0_A1_VECTOR` → TA0CCTL1.CCIFG, TA0CCTL2.CCIFG, ..., TA0CTL.TAIFG
- `TIMER1_A0_VECTOR` → TA1CCTL0.CCIFG e
- `TIMER1_A1_VECTOR` → TA1CCTL1.CCIFG, TA1CCTL2.CCIFG, ..., TA1CTL.TAIFG

Para a segunda versão desta solução, vamos usar a interrupção da passagem pelo 0 (TA0CTL.TAIFG). Neste caso, a arquitetura MSP430 compartilha um único vetor com diversas possíveis interrupções, como mostrado na Figura 7.24. A listagem abaixo apresenta uma solução, onde se nota que foi necessário consultar o registrador TA0IV.

Listagem solução do ER 7.3.b

```
// ER 7.3b
```

```

// Relógio com horas, minutos, segundos e décimos de segundos
// Usar o SMCLK = 1.048.576 Hz e interrupção com TA0CTL.TAIFG

#include <msp430.h>

#define TRUE      1
#define DEC_SEG 52428 //Contagens de SMCLK/2 para um décimo de segundo

void config_leds(void);
void isr_taifg(void);

volatile unsigned int dseg, seg, min, hora;

int main(void) {
    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer

    config_leds();
    TA0CTL = TASSEL_2 | //Selecionar SMCLK
             ID_1 | //Divisor por 2
             MC_1 | //Modo Ascendente (UP)
             TAIE; //Hab Interrupção TAIFG
    TA0CCR0 = DEC_SEG;
    __enable_interrupt(); //Hab. Geral (GIE=1)
    dseg=seg=min=hora=0;
    while(TRUE); //Programa fica parado
    return 0;
}

// Rotina de interrupção
// Acontece a cada 0,1 segundo
#pragma vector = TIMER0_A1_VECTOR
//#pragma vector = 52
__interrupt void isr_ta0(void) {
    int n;
    n = __even_in_range(TA0IV, 0xE);
    switch(n) {
        case 0x0: break; //Nenhuma interrupção pendente
        case 0x2: break; //TA0CCTL1.CCIFG
        case 0x4: break; //TA0CCTL2.CCIFG
        case 0x6: break; //TA0CCTL3.CCIFG
        case 0x8: break; //TA0CCTL4.CCIFG
        case 0xA: break; //TA0CCTL5.CCIFG (Não existe no TA0)
        case 0xC: break; //TA0CCTL6.CCIFG (Não existe no TA0)
        case 0xE: isr_taifg(); break; //TA0CTL.TAIV
    }
}

// Chamada a cada 0,1 segundo
void isr_taifg(void) {
    P4OUT ^= BIT7; //0,1 s --> Inverter led Verde
}

```

```

if (++dseg == 10) {                                //10 dec_seg?
    dseg = 0;
    P1OUT ^= BIT0;                               //1 seg --> Inverter led Vermelho
    if (++seg == 60) {                            //60 seg?
        seg = 0;
        if (++min == 60) {                      //60 min?
            min = 0;
            if (++hora == 24)      //24 horas?
                hora = 0;
        }
    }
}
void config_leds(void){...}    //Copiar do ER 7.1

```

Na solução apresentada, como o vetor 52 é compartilhado com várias *flags*, foi necessário consultar o registrador TA0IV para descobrir (usando *switch/case*) qual, dentre as candidatas, provocou a interrupção. A leitura do registrador TA0IV apaga a *flag* de interrupção de maior prioridade.

É possível fazer uma simplificação. Como temos a certeza de que habilitamos apenas uma das interrupções que compartilha o vetor 52, podemos supor que quando esta interrupção acontece, ela foi causada pela *flag* TAIFG. Porém, é preciso ler o registrador TA0IV para apagar esta *flag*, ver linha em negrito. Uma alternativa é apagar diretamente esta *flag* com TA0CTL &= ~TAIFG.

```

#pragma vector = TIMER0_A1_VECTOR
__interrupt void isr_ta0(void) {
    TA0IV;                                     //Leitura apaga flag TAIFG
    P4OUT ^= BIT7;                             //0,1 s --> Inverter led Verde
    if (++dseg == 10) {                        //10 dec_seg?
        dseg = 0;
        P1OUT ^= BIT0;                         //1 seg --> Inverter led Vermelho
        if (++seg == 60) {                      //60 seg?
            seg = 0;
            if (++min == 60) {                  //60 min?
                min = 0;
                if (++hora == 24)      //24 horas?
                    hora = 0;
            }
        }
    }
}

```

**ER 7.4.** Usando a sugestão apresentada na página 465 do manual “MSP 430 – User Guide”, (item 17.2.3.3 Use of Continuous Mode) pisque o led vermelho (P1.0) na frequência de 2 Hz e o verde (P4.7) na frequência de 3 Hz. Use o ACLK (32.768 Hz).

### Solução:

O Manual do Usuário do MSP 430 apresenta uma forma bem interessante de se gerar interrupções periódicas, com o contador operando no Modo Contínuo. Lembrar que neste modo, o contador excursiona de 0x0000 até 0xFFFF e volta a contar a partir de 0x0000, como mostrado na Figura 7.39. Esta figura ilustra o caso de se necessitar de um intervalo de tempo igual a 20.000 períodos do relógio que aciona o contador. O registrador TAxCCRn partiu com o valor 10.000. Toda vez que ocorrer a interrupção do comparador, soma-se 20.000 a esse registrador. Como ele é de 16 bits, não é necessário se preocupar com o *overflow* da soma.

Com um pouco mais de trabalho, pode-se fazer o mesmo com o contador no Modo Ascendente (*Up*). Neste caso, cada vez que se adicionar um valor ao comparador, é necessário verificar se o resultado não ultrapassou o valor de comparador 0 (TAxCCR0). Caso isso aconteça, é necessário subtrair o valor de TAxCCR0.

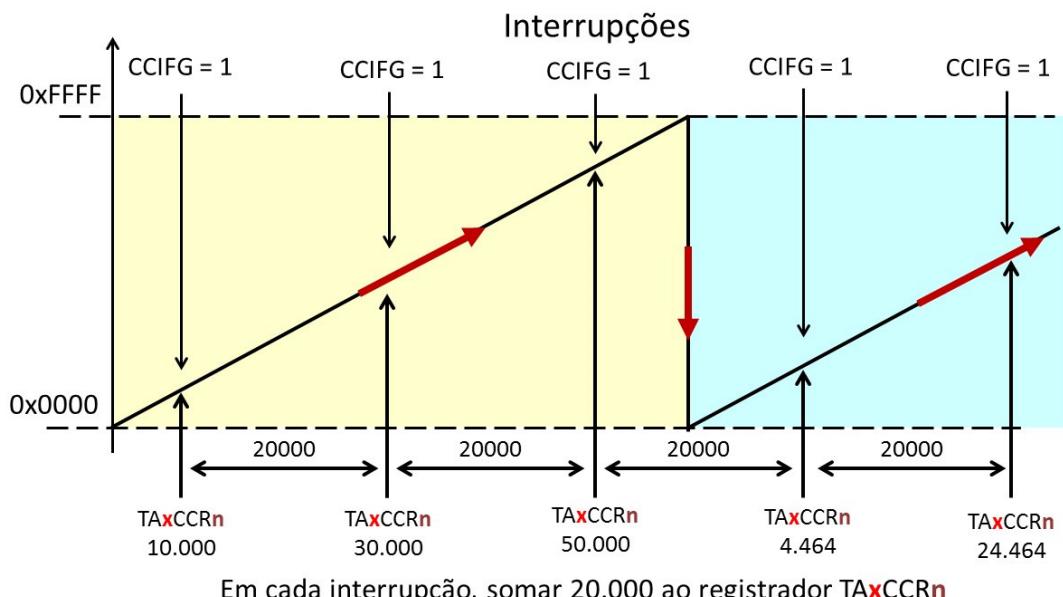


Figura 7.39. Geração de interrupções a intervalos constantes.

Vamos analisar os intervalos necessários para piscar cada led:

- **Led vermelho** → piscar em 2 Hz, isto significa 0,25 s aceso e 0,25 s apagado. O intervalo de 0,25 s corresponde a 8.192 contagens do ACLK. Será usado o comparador 1 (TA0CCR1).

- **Led verde** → piscar em 3 Hz, isto significa 0,167 s aceso e 0,167 s apagado. O intervalo de 0,167 s corresponde a 5.461 contagens do ACLK (o valor exato é 5.461,333...). Será usado o comparador 2 (TA0CCR2).

A listagem abaixo apresenta a solução. Habilitamos as interrupções e inicializamos os dois comparadores (TA0CCR1 e TA0CCR2). A cada interrupção, somamos ao comparador o valor acima calculado. Para que o programa ficasse elegante, definimos constantes com os valores a serem somados (INTERV\_2HZ e INTERV\_3HZ). O programa é simples e dispensa maiores explicações.

#### Listagem solução do ER 7.4

```
// ER 7.4
// Usar TA0 com ACLK (32.768 Hz) para piscar leds
// Vermelho: em 2 Hz, inverter estado do led a cada 250,00 ms
// Verde:      em 3 Hz, inverter estado do led a cada 166,66 ms

#include <msp430.h>

#define TRUE      1
#define INTERV_2HZ 8192    //Intervalo = 0,25 * 32768
#define INTERV_3HZ 5461    //Intervalo = 0,167 * 32768

void config_leds(void);
void isr_ccifg1(void);
void isr_ccifg2(void);

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;    // stop watchdog timer

    config_leds();
    TA0CTL = TASSEL_1 |      //Selecionar ACLK
              MC_2;          //Modo Contínuo
    TA0CCTL1 = CCIE;         //Hab int TA0CCTL1.CCIFG
    TA0CCTL2 = CCIE;         //Hab int TA0CCTL2.CCIFG
    TA0CCR1 = INTERV_2HZ;    //Inicializar comparador 1
    TA0CCR2 = INTERV_3HZ;    //Inicializar comparador 2
    __enable_interrupt();    //Hab. Geral (GIE=1)
    while(TRUE);            //Programa fica parado
    return 0;
}

// Rotina de interrupção
//#pragma vector = 52
#pragma vector = TIMER0_A1_VECTOR
__interrupt void isr_ta0(void){
```

```

int n;
n = __even_in_range(TA0IV, 0xE);
switch(n) {
    case 0x2: isr_ccifg1(); break;           //TA0CCTL1.CCIFG
    case 0x4: isr_ccifg2(); break;           //TA0CCTL2.CCIFG
    default: break;                         //Demais casos
}
}

// Inverter led Vermelho
void isr_ccifg1(void){
    P1OUT ^= BIT0;
    TA0CCR1 += INTERV_2HZ; //Próxima interrupção
}

// Inverter led Verde
void isr_ccifg2(void){
    P4OUT ^= BIT7;
    TA0CCR2 += INTERV_3HZ; //Próxima interrupção
}
void config_leds(void){...} //Copiar do ER 7.1

```

Curiosidade: Remova ou comente as duas linhas listadas abaixo. O que aconteceu? Explique o porquê.

```

TA0CCR1 = INTERV_2HZ; //Inicializar comparador 1
TA0CCR2 = INTERV_3HZ; //Inicializar comparador 2

```

**ER 7.5.** Construa um cronômetro que possa ser usado para medir o tempo consumido na execução de programas.

#### Solução:

Vamos propor um cronômetro virtual que ofereça recursos semelhantes aos de um cronômetro real. As seguintes funções abaixo irão simular esses recursos:

- void **crono\_inic** (void) → criar ou inicializar o cronômetro
- void **crono\_zera** (void) → zerar o cronômetro
- void **crono\_start** (void) → dar partida ao cronômetro
- void **crono\_stop** (void) → parar o cronômetro
- unsigned long **crono\_ler** (void) → retorna a leitura atual do cronômetro
- unsigned long **crono\_erro** (void) → mede o erro intrínseco do cronômetro

A sugestão é empregar o *Timer TA2*, porque parece que os programadores têm a mania de usar os *Timers TA0* e *TA1*. O relógio será SMCLK (1.048.576 Hz), o que vai propiciar uma precisão maior que 1  $\mu$ s (953,674  $\eta$ s). Para efeitos práticos deste exercício, vamos arredondar a precisão para 1  $\mu$ s, mesmo sabendo que cometemos um pequeno erro. A

ideia básica é concatenar o contador de TA2 (TA2R) a um contador de 16 *bits* em software. Assim, teremos um contador de 32 *bits*, como mostrado na Figura 7.40. Com esses 32 *bits* contamos até 4.294.967.295  $\mu$ s (aproximadamente), o que equivale a 1 hora, 11 minutos, 34 segundos, 967 milisegundos e 295 microsegundos. Isto parece ser uma boa excursão para nosso cronômetro.

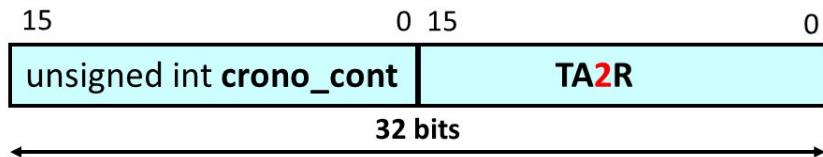


Figura 7.40. Proposta de contador de 32 *bits* para o cronômetro de acionado pelo SMCLK (1.048.576 Hz), notar o uso da variável `crono_cont`.

Portanto, a parte mais importante é o *Timer* TA2, que é colocado para operar no Modo Contínuo. Cada vez que ele volta a zero (*overflow* indicado pela *flag* TAIFG), o contador `crono_cont` é incrementado como o uso da interrupção. A listagem abaixo apresenta o pacote de funções denominado de “`crono.c`”. As diversas funções são explicadas a seguir.

```
void crono_inic(void) → Inicializa o cronômetro. Notar que é selecionado o Modo Parado (MC = 0), com isso o timer está parado. A interrupção da flag que indica a ultrapassagem (TAIFG) é habilitada com TAIE = 1. Para que a inicialização já zere o cronômetro, é ativado o bit TACLR e é zerada a variável crono_cont.
```

```
void crono_zera(void) → Simplesmente zera TA2 (TACLR) e o contador crono_cont.
```

```
void crono_start(void) → Liga o cronômetro. O contador, que estava no Modo Parado (MC = 0) muda para o Modo Contínuo (MC = 2).
```

```
void crono_stop(void) → Simplesmente volta o contador para ao Modo Parado (MC = 0). Com isso a contagem para. Com uma nova partida, o cronômetro continua a contagem do ponto em que parou.
```

```
unsigned long crono_ler(void) → Retorna em 32 bits o valor atual do cronômetro. Antes de fazer a leitura, as interrupções são desabilitadas, para evitar que o contador seja alterado durante a leitura.
```

```
unsigned long crono_erro(void) → Retorna em 32 bits o erro intrínseco do cronômetro. Teoricamente, o tempo consumido pelas funções de partida (crono_start) e de parada (crono_stop) deveria ser igual a zero. Mas isto não acontece porque essas funções consomem tempo. O erro intrínseco é calculado em 16 partidas e paradas sequenciais. Este erro deveria ser subtraído das leituras do cronômetro. É claro que essa
```

medida é bastante imprecisa e pode ser prejudicada se outras interrupções estiverem habilitadas.

`_interrupt void isr_ta2ifg(void)` → Esta é a função que atende à interrupção TAIFG. Como somente esta interrupção foi habilitada, a função ficou bem simples. A linha TA2IV faz a leitura deste registrador e assim zera a *flag* TAIFG.

### Listagem solução do ER 7.5

```
// ER 7.5 - Cronômetro com TA2
// Usa o TA2 e interrupção TAIFG

// crono_inic()    --> inicializar cronômetro
// crono_cal()     --> calibrar cronômetro
// crono_start()   --> disparar cronômetro
// crono_stop()    --> parar cronômetro
// crono_zera()    --> zerar cronômetro
// crono_ler()     --> retorna valor instantâneo

#include <msp430.h>
#include "crono.h"

// Inicializar cronometro SMCLK (1.048.576 Hz)
void crono_inic(void) {
    TA2EX0 = 0;
    TA2CTL = TASSEL_2 | TACLR | MC_0 | TAIE;      //Configurar TA2, MC=0
    crono_cont=0;                                //Zerar contador
}

// Zerar cronometro
void crono_zera(void) {
    TA2CTL |= TACLR;        //Zerar TA2
    crono_cont=0;          //Zerar TA2
}

// Disparar cronometro
void crono_start(void) {
    TA2CTL |= MC_2;        //MC=2 para ligar TA2
}

// Parar cronometro
void crono_stop(void) {
    TA2CTL &= ~MC_2;       //MC=0 para desligar TA2
}

// Ler valor cronometro
unsigned long crono_ler(void) {
    unsigned long aux;
```

```

__disable_interrupt();
aux=((unsigned long)crono_cont<<16) | TA2R;
__enable_interrupt();
return aux;
}

// Determinar erro entre Start/Stop
unsigned long crono_erro(void){
    unsigned char i;
    unsigned long tp;
    crono_zera();
    for (i=0; i<16; i++) {          // Executar 16 vezes
        crono_start();
        crono_stop();
    }
    tp = crono_ler();              // Ler valor acumulador
    return tp>>4;                // Dividir por 16
}

// Interrupção
#pragma vector = 43      // TIMER2_A1_VECTOR
__interrupt void isr_ta2ifg(void){
    TA2IV;                      // Ler TA2IV para apagar TAIFG
    crono_cont++;                // Incrementar contador
}

```

O arquivo header (crono.h) deve ser incluído no diretório do projeto que estiver usando o cronômetro. Ele está listado abaixo. É bastante simples. Apenas traz a declaração da variável global `crono_cont` e o protótipo das funções do cronômetro.

Listagem do *header* a ser usado na solução do ER 7.5

```

// crono.h

#ifndef CRONO_H_
#define CRONO_H_

unsigned int crono_cont;

void crono_inic(void);
void crono_zera(void);
void crono_start(void);
void crono_stop(void);
unsigned long crono_ler(void);
unsigned long crono_erro(void);

#endif /* CRONO_H_ */

```

Para finalizar, na listagem abaixo, apresentamos um exemplo de uso do cronômetro. Vemos que esse programa usa a função `crono_erro()` para medir o erro intrínseco do cronômetro e depois faz uma série de medições. Após rodar o programa, o usuário deve colocar o programa em pausa e usar o CCS para verificar o conteúdo das variáveis. Em nosso programa, encontramos:

`erro = 13 µs` → erro intrínseco do cronômetro;  
`tp1 = 21 µs` → tempo gasto para incrementar uma variável inteira;  
`tp2 = 114 µs` → tempo gasto para incrementar uma variável float e  
`tp3 = 10.46 µs` → tempo gasto num laço vazio com 1.000 repetições.

#### Listagem com exemplo de uso da solução do ER 7.5

```
// ER 7.5 - Exemplo de uso
// Exemplo de uso do cronômetro (TA2)

#include <msp430.h>
#include "crono.h"

int main(void)
{
    volatile int i=0;
    volatile float x=0;
    volatile unsigned long tp1,tp2,tp3,tp4,erro;
    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer

    crono_inic();
    erro=crono_erro();

    // Incremento variável "int"
    crono_zera();
    crono_start();
    i++;
    crono_stop;
    tp1=crono_ler();

    // Incremento variável "float"
    crono_zera();
    crono_start();
    x++;
    crono_stop;
    tp2=crono_ler();

    // Laço vazio com variável "int"
    crono_zera();
    crono_start();
    for (i=0; i<1000; i++) ;
}
```

```

    crono_stop;
    tp3=crono_ler();

    while(1);      //Parar num laço infinito
    return 0;
}

```

**ER 7.6.** Usando o *timer* TA0 acionado pelo SMCLK (1.048.576 Hz), faça os *leds* piscarem da seguinte forma: vermelho (P1.0) em 1 Hz e verde (P4.7) em 2 Hz.

**Solução:**

A figura abaixo, apresenta o que significa um *led* piscando em 1 Hz e em 2 Hz. Note que piscar em 1 Hz significa que o *led* deve acender e apagar dentro do período de 1 s, ou seja, 0,5 s aceso e 0,5 s apagado. De forma análoga acontece com o piscar em 2 Hz.

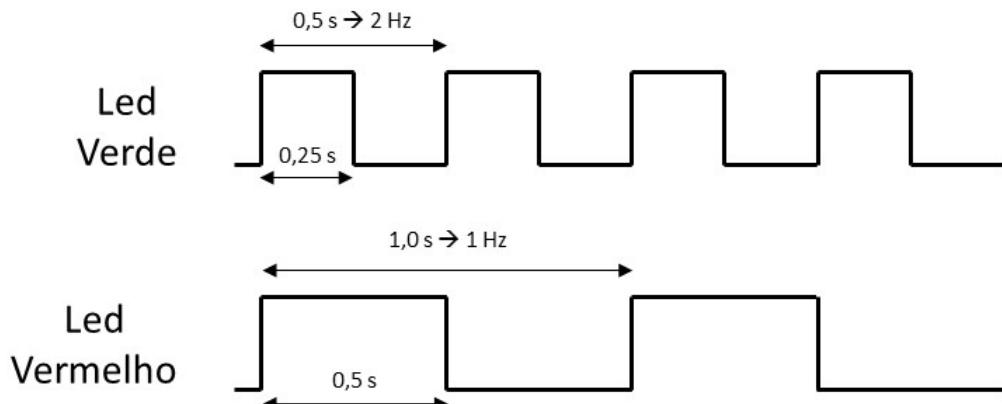


Figura 7.41. Indicação para piscar os leds nas frequências de 1 Hz e 2 Hz.

Solucionamos este problema preparando o *timer* TA0 para gerar 0,25 s. A forma mais simples é escolher o Modo Ascendente (MC = 1) e limitar a excursão da contagem com o registrador TA0CCR0, vide Figura 7.13.

Um problema surge com a imposição de se usar o SMCLK. Com este relógio, em 0,25 s temos 262.144 contagens, o que é valor acima do que se pode escrever num registrador de 16 bits. Precisamos então dividir o relógio SMCLK. Uma solução é dividi-lo por 8 (ID = 3), assim a frequência passa a ser de 131.072 Hz. Nesta situação, em 0,25 s, temos 32.768 contagens.

Vamos apresentar duas soluções. Na primeira solução ficaremos consultando (*polling*) a flag CCIFG, como mostrado na listagem abaixo. Note que invertermos o *led* verde cada vez em que essa flag for para 1 e o *led* vermelho a cada 2 vezes. A variável fase foi usada

para controlar o instante em que se inverte o *led* vermelho. Nesta solução a CPU precisa monitorar constantemente a *flag* CCIFG.

### Listagem da solução “a” do ER 7.6

```
//ER 7.6a
// Piscar led Verde em 2 Hz e
// Piscar led Vermelho em 1 Hz
// Polling TA0CCTL0.CCIFG

#include <msp430.h>

#define TRUE 1
#define FALSE 0

void config_leds(void);

int main(void){
    int fase=0;      //Fase para o led vermelho (mais lento)
    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer
    config_leds();

    TA0CTL = TASSEL_2 |      //Selecionar SMCLK (1.048.576)
             MC_1       |      //Modo 1 = ascendente
             ID_3;        //SMCLK / 8 = 131.072
    TA0CCR0 = 32767;        //(131.075 x 0,25) - 1

    while(TRUE) {
        while ( (TA0CCTL0&CCIFG) == 0); //Esperar CCIFG
        TA0CCTL0 &= ~CCIFG;           //Fazer CCIFG=0
        P4OUT ^= BIT7;              //Inverter Verde
        fase++;
        if (fase==2){               //A cada 2 passadas
            fase = 0;
            P1OUT ^= BIT0;          //Inverter Vermelho
        }
    }
    return 0;
}
void config_leds(void){...} //Copiar do ER 7.1
```

Para a segunda solução faremos uso da interrupção da *flag* CCIFG, como mostrado na listagem abaixo. Note que agora a CPU ficou liberada para outras tarefas, o que está caracterizado pelo laço infinito construído com o `while (TRUE)`. Agora tudo acontece na rotina de interrupção do comparador 0. Vale lembrar que essa interrupção zera

automaticamente a *flag* CCIFG. Na rotina de interrupção, para ficar mais elegante, apenas invertemos o *bit* 0 da variável fase (`fase ^= 1`) para acertar o pisca do *led* vermelho.

### Listagem da solução “b” do ER 7.6

```
//ER 7.6b
// Piscar led Verde em 2 Hz e
// Piscar led Vermelho em 1 Hz
// Interrupção TA0CCTL0.CCIFG

#include <msp430.h>

#define TRUE 1
#define FALSE 0

void config_leds(void);

volatile int fase=0;      //Fase para o led vermelho (mais lento)

int main(void){
    WDTCTL = WDTPW | WDTHOLD;    // stop watchdog timer
    config_leds();

    TA0CTL = TASSEL_2 |          //Selecionar SMCLK (1.048.576)
             MC_1 |           //Modo 1 = ascendente
             ID_3;            //SMCLK/8 = 131.072
    TA0CCTL0 = CCIE;           // Habilita interrupção CCIFG
    TA0CCR0 = 32767;           //(131.075 x 0,25) - 1
    __enable_interrupt();       //Habilitação geral (GIE=1)

    while(TRUE);                //Laço infinito

    return 0;
}

//Rotina para tratar interrupção TA0CCTL0.CCIFG
#pragma vector = 53
__interrupt void isr_ccifg0(void){
    P4OUT ^= BIT7;              //Inverter Verde
    fase ^= 1;                  //Alternar fase entre 0 e 1
    if (fase==1)
        P1OUT ^= BIT0;          //Inverter Vermelho
}
void config_leds(void){...}   //Copiar do ER 7.1
```

**ER 7.7.** Este e os próximos dois exercícios estão relacionados, pois usam o gerador PWM para controlar o brilho dos *leds*. Para começar, este exercício pede que se altere de forma cíclica o brilho do *led* vermelho usando os valores de 10%, 50% e 80%, usando o recurso

de PWM do *Timer TA0.1* (com ACLK), na frequência de 50 Hz (período = 20 ms). Com um cabo fêmea-fêmea, conecte o pino P1.2 (TA0.1) ao pino 2 do jumper 8 da placa de ensaios, assim, a saída do comparador 1 do *Timer TA0* passa a acionar o *led* vermelho, vide Figura 7.42.

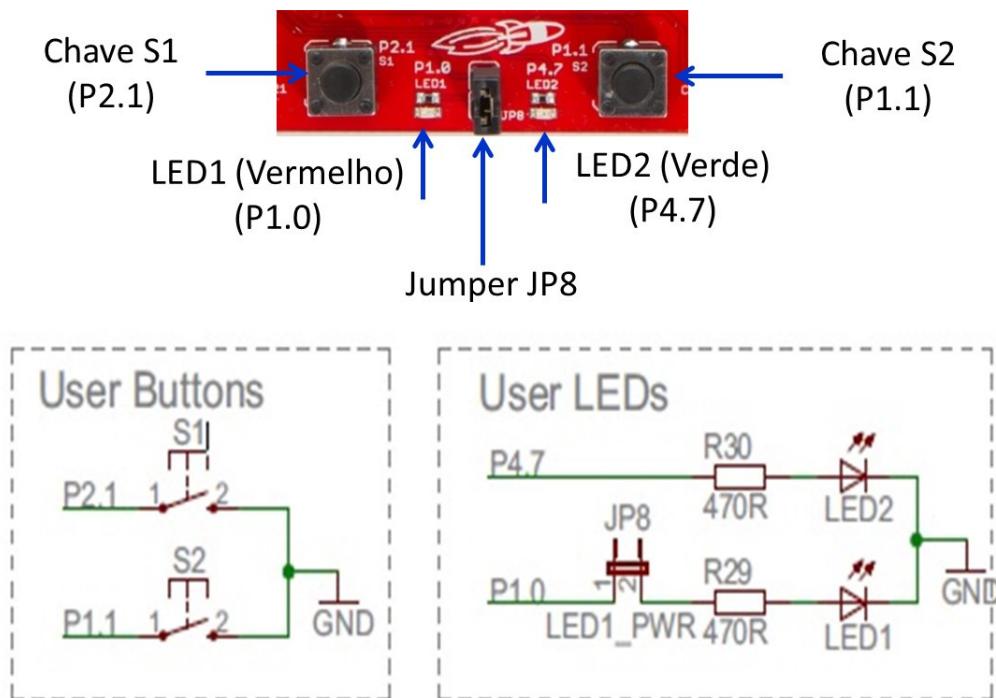


Figura 7.42. Esquema da conexão das chaves (S1 e S2) e dos leds (LED1 e LED2) disponíveis na placa de ensaios (LaunchPad).

**Solução:** Começamos indicando que o período do PWM é um parâmetro importante e seu valor depende da aplicação. Como o caso aqui é controlar o brilho de um *led*, vamos lembrar da chamada Persistência da Visão. Em geral, o olho humano tem a sensação de continuidade de movimento se a exposição de imagens for acima de 16 Hz. O cinema usa 24 quadros por segundo (24 Hz). Resta o problema da Cintilação, pois distinguimos uma luz piscando até a frequência de 30 Hz. O cinema usa 48 Hz, pois cada quadro é exposto duas vezes. Assim, para o caso de nosso *led*, o pedido orienta para usar 50 Hz (período de 20 ms), o que é suficiente para evitar a cintilação e teremos a sensação de que o *led* está aceso. Vale lembrar que a visão periférica é capaz de perceber cintilação até próxima de 60 Hz, mas não vamos levar isso em conta.

De forma bem simples, todo problema de PWM se resume a duas perguntas:

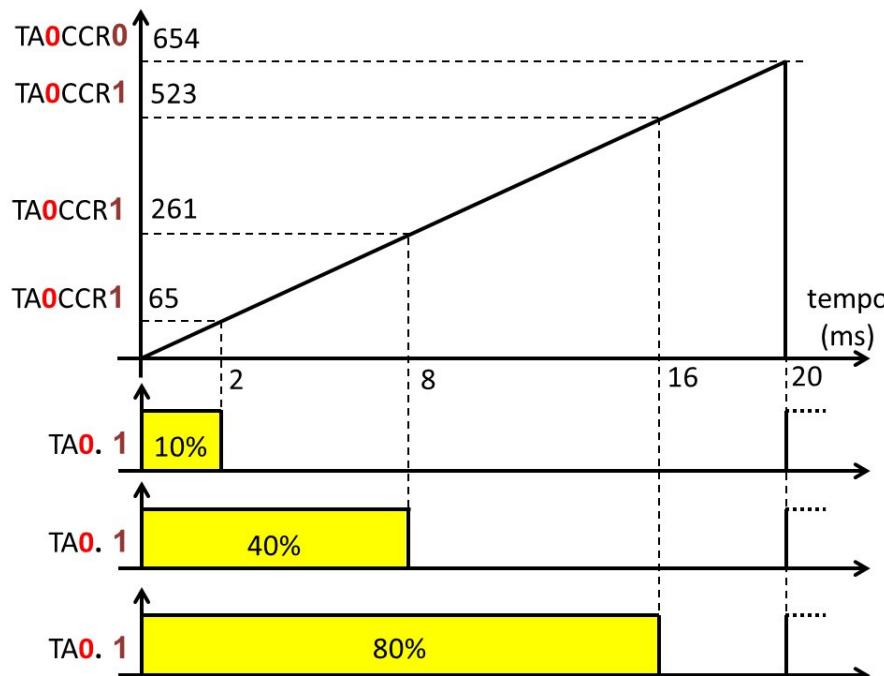
- 1) Qual o período do PWM?
- 2) Qual o Ciclo de Carga (*Duty Cycle*)?

Para o atual problema usando o ACLK, temos:

- 1) O período do PWM é de 20 ms, o que corresponde a 655 contagens do ACLK e
- 2) São pedidos 3 ciclos de carga:      10% → 66 contagens ( $655 \times 0,1$ );  
 50% → 328 contagens ( $655 \times 0,5$ ) e  
 80% → 534 contagens ( $655 \times 0,8$ ).

Este problema, que pede um período relativamente grande (20 ms), impôs o uso do ACLK (32.768 Hz) para acionar o TA0. Vamos escolher o Modo Ascendente (MC = 1). Para garantir o período do PWM e programar TA0CCR0 para que o contador TA0R complete um ciclo de contagem a cada 20 ms. Precisaremos então de aproximadamente de 655 contagens ( $32.768 \times 0,20 = 655,36$ ), por isso faremos TA0CCR0 = 654.

O problema pede para gerar o PWM usando TA0.1, ou seja, usando a Unidade 1 de Comparação (TA0CCR1). Vamos programá-la de forma a habilitar sua Unidade Geradora de Saída (OUT). A Figura 7.19 apresenta as opções de operação desta unidade. Por enquanto, a opção mais interessante é o Modo 6, na qual o ciclo de carga é diretamente proporcional ao valor do registrador de comparação. A Figura 7.43 ilustra este caso.



*Figura 7.43. Ilustração do uso de TA0.1 para gerar PWM com período de 20 ms e com três Ciclos de Carga (10%, 40% e 80%).*

A listagem abaixo apresenta o programa solução. Ele é bem simples e dispensa maiores explicações. Para que o leitor pudesse ver os diferentes ciclos de carga, o programa faz uma pausa de 2 s com o uso da função `__delay_cycles()`.

#### Listagem da solução do ER 7.7

```
//ER 7.7
// Controlar brilho do LED1 com PWM (TA0.1)
// Período de 20 ms, Ciclos de Carga 20%, 50% e 80%
// Usar cabo para ligar pino P1.2 (TA0.1) ao pino 2 do Jumper JP8

#include <msp430.h>

void config_ta01(void);
void config_p12(void);

#define TRUE 1

#define T20ms 654          //período de 20 ms, (32768 x 0,02) - 1
#define PWM10 (0.1*T20ms) //10% de ciclo de carga
#define PWM40 (0.4*T20ms) //40% de ciclo de carga
#define PWM80 (0.8*T20ms) //80% de ciclo de carga

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;      // stop watchdog timer
    config_p12();
    config_ta01();

    while(TRUE) {
        TA0CCR1 = PWM10;           //10%
        __delay_cycles(2000000);   //Espera 2 segundos
        TA0CCR1 = PWM50;           //40%
        __delay_cycles(2000000);   //Espera 2 segundos
        TA0CCR1 = PWM80;           //80%
        __delay_cycles(2000000);   //Espera 2 segundos
    }
    return 0;
}

// Configurar TA0.1
void config_ta01(void) {
    TA0CTL = TASSEL_1 | MC_1;    //TA0 com ACLK e modo 1
    TA0CCR0 = T20ms;            //Limite para 20ms
    TA0CCTL1 = OUTMOD_6;        //TA0.1, saída no Modo 6
}

// Configurar P1.2 (TA0.1)
void config_p12(void) {
```

```

P1DIR |= BIT2; //P1.2 como saída
P1SEL |= BIT2; //P1.2 dedicado ao TA0.1
}

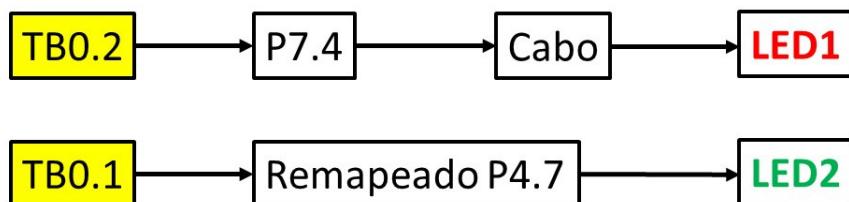
```

**ER 7.8.** Com o *Timer* B0 acionado pelo SMCLK (1.048.576 Hz), configure um PWM com período de 20 ms (50 Hz) para controlar o brilho dos dois *leds* (*led1* = vermelho e *led2* = verde), excursionando de forma complementar, de 0% até 100% em passos de 10%. As chaves S1 e S2 serão usadas neste controle, operando da forma mostrada na Tabela 7.16. Note que por acionamento de uma chave se entende sua passagem do estado de aberta para o estado de fechada.

*Tabela 7.16. Comportamento das chaves para variar o Ciclo de Carga do PWM que aciona os leds*

Chave	LED1 (vermelho)	LED2 (verde)
S1	+10%	-10%
S2	-10%	+10%

Neste problema serão usadas as instâncias 0 e 1 dos comparadores do *Timer* B0 (TB0.1 e TB0.2), para acionar os *leds* de acordo com a conexão na figura abaixo. Reveja o trecho do Capítulo 3 (item 3.5) que trata do mapeamento da Porta P4.



*Figura 7.44. Esquema usado para que o Timer B0 acione os dois leds.*

**Solução:** Vamos operar com o TB0 no Modo 1, assim em TB0CCR0 configuramos o período do PWM.

Período de 20 ms:  $0,020 \times 1.048.576 = 20.971,52 \rightarrow \text{TB0CCR0} = 20.971$  contagens.

O valor 20.971,5 deveria ser arredondado para 20.972, mas precisamos subtrair 1 porque o zero também é contado. Com esse valor, o erro no período de 20 ms é de 5 ns, o que é bastante aceitável.

Para facilitar o controle da potência, vamos usar duas variáveis, `pot1` e `pot2`, que contando de 0 até 100, vão guardar a potência atual de cada `led`. De acordo com o acionamento das chaves somamos ou subtraímos 10 de cada variável. Há que se tomar cuidado para essas variáveis não irem acima de 100% ou abaixo de 0%. O monitoramento das chaves não é objetivo deste capítulo. O leitor deve consultar o Capítulo 5. A conta para calcular o valor dos comparadores é mostrada a seguir.

```
TB0CCR1 = (POT100 * (long)pot2) / 100; //Programar PWM
TB0CCR2 = (POT100 * (long)pot1) / 100; //Programar PWM
```

Um outro ponto que precisa ser citado é sobre a sensibilidade de nosso olho às variações de luminosidade. Essa sensibilidade não é linear, o leitor vai notar que percebe bem as variações de brilho de quando há pouca potência, mas que pouca alteração percebe quando o brilho está acima de 70%. A listagem do programa solução é apresentada a seguir.

#### Listagem da solução do ER 7.8

```
//ER 7.8
// Controlar brilho dos dois leds com PWM do TB0
// Chave S1 (P2.1) ==> Vermelho (+10%) e Verde (-10%)
// Chave S2 (P1.1) ==> Verde (+10%) e Vermelho (+10%)
//
// TB0.2 --> P7.4 --> Cabo --> Vermelho (Led 1)
// TB0.1 --> mapeamento P4.7 --> Verde (Led 2)

#include <msp430.h>

#define TRUE    1
#define FALSE   0

#define ABT 1      //Constance representa Aberta
#define FEC 0      //Constance representa Fechada
#define DBC 1000   //Atraso para o debounce

#define POT100 20971 //0,02 * 1.048.576 --> 10 ms

void config_tb0(void);           //Config TB0
void config_pinos(void);         //Config Pinos
int check_s1(void);              //Checar chave S1
int check_s2(void);              //Checar chave S2
void debounce(void);             //Delay para debounce

int ps1=ABT,ps2=ABT;            //Estado anterior das chaves

int main(void)
```

```

{
    volatile int pot1=50,pot2=50;      //Iniciar com 50%
    WDTCTL = WDTPW | WDTHOLD;        // stop watchdog timer
    config_pinos();                  //Config pinos
    config_tb0();                   //Config TB0
    while (TRUE){
        if (check_s1()){           //Chave S1?
            pot1+=10;             //pot1 = pot1 + 10%
            pot2-=10;             //pot2 = pot2 - 10%
            if (pot1 > 100) pot1=100; //Limite 100%
            if (pot2 < 0)   pot2=0;  //Limite 0%
        }
        if (check_s2()){
            pot1-=10;             //pot1 = pot1 - 10%
            pot2+=10;             //pot2 = pot2 + 10%
            if (pot1 < 0)   pot1=0;  //Limite 0%
            if (pot2 > 100) pot2=100; //Limite 100%
        }
        TB0CCR1 = (POT100 * (long)pot2) / 100; //Programar PWM
        TB0CCR2 = (POT100 * (long)pot1) / 100; //Programar PWM
    }
    return 0;
}

// Configurar TB0
void config_tb0(void){
    TB0CTL = TBSSEL_2 | MC_1;      //TB0 com SMCLK e Modo Up
    TB0CCR0 = POT100;              //Período PWM
TB0CCTL1 = OUTMOD_6;          //Saída Modo 6
TB0CCTL2 = OUTMOD_6;          //Saída Modo 6
    TB0CCR1 = POT100/2;            //Iniciar com 50%
    TB0CCR2 = POT100/2;            //Iniciar com 50%
}

// Verificar S1
// Retorna: TRUE = A->F, FALSE = demais casos
int check_s1(void){
    if ((P2IN&BIT1) == 0){ //S1 fechada
        if (ps1==ABT){        //A --> F
            debounce();
            ps1=FEC;           //Guardar o passado
            return TRUE;
        }
        else
            return FALSE;     //F --> F
    }
    else{                      //S1 aberta
        if (ps1==ABT)
            return FALSE;     //A --> A
        else{

```

```

        debounce();
        ps1=ABT;           //Guardar o passado
        return FALSE;      //F --> A
    }
}

// Verificar S2
// Retorna: TRUE = A->F, FALSE = demais casos
int check_s2(void){
    if ((P1IN&BIT1) == 0){ //S1 fechada
        if (ps2==ABT){ //A --> F
            debounce();
            ps2=FEC;       //Guardar o passado
            return TRUE;
        }
        else
            return FALSE; //F --> F
    }
    else{                //S1 aberta
        if (ps2==ABT)
            return FALSE; //A --> A
        else{
            debounce();
            ps2=ABT;       //Guardar o passado
            return FALSE; //F --> A
        }
    }
}

void config_pinos(void){
    P2DIR &= ~BIT1;      //S1 = entrada com pull-up
    P2REN |= BIT1;
    P2OUT |= BIT1;

    P1DIR &= ~BIT1;      //S2 = entrada com pull-up
    P1REN |= BIT1;
    P1OUT |= BIT1;

    P7DIR |= BIT4;       //TB0.2 = P7.4
    P7SEL |= BIT4;       //Ligar cabo para Led Vermelho

    P4DIR |= BIT7;
    P4SEL |= BIT7;
    PMAPKEYID = 0X02D52; //Escrever chave
    P4MAP7 = PM_TB0CCR1A; //TB0.1 mapeado para P4.7
}

// Rotina de atraso para o debounce das chaves

```

```
void debounce(void) {
    volatile unsigned int i;
    for (i=DBC; i>0; i--);
}
```

**Desafio:** Ao rodar o programa, o leitor vai constatar seu correto funcionamento, exceto que de, de vez em quando, um dos *leds* dá uma piscada mais forte. Por que será que isso acontece? Substitua as duas linhas do programa que estão em negrito, pelas duas linhas abaixo. Agora, esse pisca ocasional vai sumir. Por quê?

```
TB0CCTL1 = CLLD_1 | OUTMOD_6;
TB0CCTL2 = CLLD_1 | OUTMOD_6;
```

**ER 7.9.** Este exercício é muito semelhante ao anterior, mas sem usar as chaves. O programa dever fazer, de forma complementar, o brilho dos *leds* ir de 0% até 100% em passos de 1%. A razão do incremento é de 10 passos de 1% a cada 1 segundo. Então, de forma cíclica, enquanto um *led* vai de 0% até 100%, o outro vai de 100% até 0%.

**Solução:** Poderíamos usar a mesma solução do exercício anterior e criar uma variável para ir contando a potência e, a cada novo valor, calcular as atualizações para TB0CCR1 e TB0CCR2. Porém, vamos usar uma solução mais simples. Pelos cálculos do exercício anterior, o período do PWM corresponde a 20.972 contagens do *timer* e, portanto, 1% deste valor resulta em 209,72. Arredondamos para 210.

Se consideramos que 1% corresponde a 210 contagens então 100% corresponde a 21.000 contagens. No período que foi especificado para 20 ms, teremos um erro de:

$$erro = 0,020 - \frac{21.001}{1.048.576} = 28\mu s$$

No exercício anterior usamos o contador no Modo Ascendente e garantimos o período com o valor do comparador TB0CCR0. Por isso, a coincidência com TB0CCR0 acontece na taxa de 50 Hz, ou seja, 50 por segundo. Vamos então habilitar essa interrupção e, a cada 5 interrupções, incrementamos ou decrementamos os comparadores que controlam o ciclo de carga dos PWMs. A variável fase vai controlar o sentido dos incrementos e decrementos. Para facilitar o raciocínio, vamos tomar como referência a contagem feita para o comparador 1.

- fase = 0 → decrementar TB0CCR1 ( $\uparrow led$  1 = verde e  $\downarrow led$  2 = vermelho) e
- fase = 1 → incrementar TB0CCR1 ( $\downarrow led$  1 = verde e  $\uparrow led$  2 = vermelho).

Listagem da solução do ER 7.9

```

// ER 7.9
// TB0 para gerar PWM e controlar brilho dos leds
// Passo automático de 1% a cada 100 mseg
//
// TB0.2 --> P7.4 --> Cabo --> Vermelho (Led 1)
// TB0.1 --> mapeamento P4.7 --> Verde (Led 2)

#include <msp430.h>

#define TRUE      1
#define FALSE     0

#define PASSO    210          //0,0002 * 1.048.576 --> 200 useg
#define POT100   (100*PASSO)  //Potência = 100%

void config_tb0(void);        //Config TB0
void config_pinos(void);      //Config Leds

volatile int cont;            //Contar interrupções
volatile int fase;            //Sentido do incremento de 1% (TB0CCR1)
                            // fase=0 → (led2) TB0CCR1 += PASSO
                            // fase=1 → (led1) TB0CCR1 -= PASSO

int main(void){
    WDTCTL = WDTPW | WDTHOLD;    //stop watchdog timer
    config_pinos();              //Config pinos
    config_tb0();                //Config TB0
    __enable_interrupt();         //Habilitação geral (GIE=1)
    while (TRUE);                //Laço infinito
    return 0;
}

// Configurar TB0
void config_tb0(void){
    TB0CTL = TBSSEL_2 | MC_1;    //TB0 com SMCLK e Modo Up
    TB0CCR0 = POT100;           //Período PWM
    TB0CCTL0 = CCIE;            // Interrup TB0CCR0 CCIFG (vetor 59)
TB0CCTL1 = CLLD_1 | OUTMOD_6; //Saída Modo 6
TB0CCTL2 = CLLD_1 | OUTMOD_6; //Saída Modo 6
    TB0CCR1 = POT100;           //Led2 = 100%
    TB0CCR2 = 0;                //Led1 = 0%
    fase=0;                     //Descer potência TB0CCR1 (led2)
}

void config_pinos(void){
    P7DIR |= BIT4;             //TB0.2 = P7.4
    P7SEL |= BIT4;              //Ligar cabo para Led Vermelho

    P4DIR |= BIT7;
}

```

```

P4SEL |= BIT7;
PMAPKEYID = 0X02D52;      //Escrever chave
P4MAP7 = PM_TB0CCR1A;    //TB0.1 mapeado para P4.7
}

//Rotina para tratar interrupção TB0CCTL0.CCIFG
//#pragma vector = TIMER_B0_VECTOR
#pragma vector = 59
__interrupt void isr_tb0_ccifg0(void) {
    if (++cont == 5){ //Contar 10 mseg
        cont=0;
        if (fase == 1){ //Sobe led1 (CCR2) e desce led2 (CCR1)
            if (TB0CCR1 == POT100){ //Led2=100%?
                fase=0;           //Mudar sentido
                TB0CCR1 -= PASSO;
                TB0CCR2 = PASSO;
            }
            else{
                TB0CCR1 += PASSO;
                TB0CCR2 -= PASSO;
            }
        }
        else{ //Desce led1 e sobe led2
            if (TB0CCR1 == 0){ //Led2=0%?
                fase=1;           //Mudar sentido
                TB0CCR1 = PASSO;
                TB0CCR2 -= PASSO;
            }
            else{
                TB0CCR1 -= PASSO;
                TB0CCR2 += PASSO;
            }
        }
    }
}

```

Desafio: Na listagem acima, o leitor é convidado a substituir as duas linhas em negrito, pelas duas linhas abaixo. O que notou de diferente? Consegue explicar?

```
TB0CCTL1 = OUTMOD_6; //Saída Modo 6  
TB0CCTL2 = OUTMOD_6; //Saída Modo 6
```

**ER 7.10.** Construa um programa capaz de gerar em P2.5 (TA2.2 com SMCLK) uma onda quadrada nas frequências das notas musicais durante períodos específicos e assim poder tocar pequenos trechos de músicas usando como transdutor um *buzzer* passivo. Também

pode ser usado um amplificador de áudio (GF1002) e um pequeno alto falante. Veja figura no EP 7.8.

**Solução:** Pensando numa solução para “tocar” músicas, podemos imaginar que a função básica seria uma capaz de gerar na saída a onda quadrada na frequência solicitada, durante o tempo especificado. Para gerar onda quadrada, vamos usar o recurso de PWM, sendo que o período do PWM garante a frequência solicitada e seu ciclo de carga será sempre programado em 50%, para gerar onda quadrada. Para isso, vamos configurar o contador (TA2R) para o Modo Ascendente (Up) e usar TA2CCR0 para limitar a contagem de acordo com o período especificado. Sugerimos então a função cujo protótipo está abaixo, onde `freq` é a frequência em Hz e `tp` é a duração do tom, em milissegundos.

```
void tom (int freq, int tp);
```

Então, essa função deve programar TA2.2 para gerar na frequência especificada um PWM com 50% de carga e retornar após decorrido o parâmetro `tp`, que está em milissegundos. A forma de contar o período `tp` pode ser bem simples se for usada como referência a frequência do PWM.

Por exemplo, vamos considerar a nota Dó (261 Hz na 4<sup>a</sup> oitava) sendo gerada por 500 ms. Nesse caso, vamos programar o período do PWM (1/261 s) e, a cada segundo, teremos 261 coincidências com TA2CCR0. Se queremos 500 ms, basta esperarmos 130 (261/2) coincidências. A listagem abaixo apresenta uma sugestão para esta função. Note que o cálculo para atualizar a variável `cont` é feito em ponto flutuante pois usou “1000.0”.

```
// Gerar tom na frequência "freq" durante "tp" milisegundos
void tom(int freq, int tp){
    int cont;
    TA2CCR0 = 1048576L/freq;           //Programar o período
    TA2CCR2 = TA2CCR0/2;               //Carga de 50%
    cont = freq*(tp/1000.0);          //Calcular qtd de coincidências
    while(cont > 0){
        while( (TA2CCTL0&CCIFG) == 0); //Esperar TA2CCR0.CCIFG
        TA2CCTL0 &= ~CCIFG;            //Apagar flag
        cont--;                      //Decrementar contador
    }
}
```

Bem, já temos o básico. Agora precisamos de uma tabela com a frequência de cada nota musical. A Tabela 7.17 foi conseguida com uma busca na Internet. Nesta tabela, a letra “S” indica sustenido e a letra “H” indica a próxima escala (H = high). A cifra que seria “C” foi alterada para “CC”, pois a letra “C” parecer ser reservada pelo Code Composer. Para facilitar a “programação” da música vamos numerar essa sequência. Sob a coluna

“Índice”, cada nota recebeu um número. Em linguagem “C” isto é feito com facilidade usando o recurso de enumeração. Note que adicionamos a nota ZR (lembra zero) para indicar frequência zero, ou seja, períodos de pausa.

```
enum notas {ZR, CC, CS, D, DS, E, . . . , GSH, AH, ASH, BH};
```

*Tabela 7.17. Tabela com as notas musicais e suas frequências*

<b>Oitava 4</b>			
<b>Notas</b>	<b>Índice</b>	<b>Cifra</b>	<b>Hz</b>
Dó	1	CC	261
Sust.	2	CS	277
Ré	3	D	293
	4	DS	311
Mi	5	E	329
Fá	6	F	349
	7	FS	369
Sol	8	G	392
	9	GS	415
Lá	10	A	440
	11	AS	466
Si	12	B	493

<b>Oitava 5</b>			
<b>Notas</b>	<b>Índice</b>	<b>Cifra</b>	<b>Hz</b>
Dó	13	CH	523
Sust.	14	CSH	554
Ré	15	DH	587
	16	DSH	622
Mi	17	EH	659
Fá	18	FH	698
	19	FSH	739
Sol	20	GH	784
	21	GSH	830
Lá	22	AH	880
	23	ASH	932
Si	24	BH	987

Tendo pronta a função tom (...) e a tabela com as notas, a música a ser tocada se resume em um vetor de notas e cada nota com sua duração. Para construir tal vetor é necessário um certo conhecimento musical. O exemplo deste exercício se baseou na sugestão feita para o Arduino que está disponível em:

//<https://dragaosemchama.com/2014/04/player-de-marcha-imperial-8-bitsarduino.>

A listagem abaixo apresenta o programa solução. Note que estão marcados a enumeração das notas musicais, a definição da frequência de cada nota e, finalmente, o vetor com a Marcha Imperial. Este vetor é composto por pares: nota, duração. A função tom(...) está ampliada pois foi necessário introduzir as pausas (nota = ZR). O final do vetor marcha[] está marcado com o par ZR, ZR. O programa principal, quando encontra este par, volta a repetir a sequência.

#### Listagem da solução do ER 7.10

```
// ER 7.10
// Programa para tocar músicas
// Exemplo com a Marcha Imperial do Star Wars

// Usar um buzzer passivo (sem oscilador interno) tipo KX-1212
```

```

// Cuidado: a maioria dos buzzers à venda tem oscilador interno

// Ligar o "+" no pino P2.5
// Ligar o "-" no GND

// Inspirado em:
//https://dragaosemchama.com/2014/04/player-de-marcha-imperial-8-bitsarduino/
//http://www.musicnotes.com/sheetmusic/mtd.asp?ppn=MN0016254

#include <msp430.h>

#define TRUE 1

/////////////////////////////// Definição das notas e frequências //////////////////
/////////////////////////////// Criar constantes: ZR=0, CC=1, CS=2, D=3, ..., BH=24
enum notas {ZR, CC, CS, D, DS, E, F, FS, G, GS, A, AS, B, CH,
            CSH, DH, DSH, EH, FH, FSH, GH, GSH, AH, ASH, BH};

// Vetor com as notas musicais
// "S" para sostenido e "H" para indicar a escala superior
int freq[]={
    //ZR CC CH D DH E F FH G GH A AH B
    0, 261, 277, 293, 311, 329, 349, 369, 392, 415, 440, 466, 493,
    //CS CSH DS DSH ES FS FSH GS GSH AS ASH BS
    523, 554, 587, 622, 659, 698, 739, 784, 830, 880, 932, 987};

/////////////////////////// Marcha Imperial /// Final marcado com ZR,ZR //////////////////
/////////////////////////////// Marcha Imperial /// Final marcado com ZR,ZR //////////////////
int marcha[]={
    A, 500, ZR, 100, A, 500, ZR, 100, A, 500, ZR, 100, F, 350,
    CH, 150, A, 500, F, 350, CH, 150, A, 1000, EH, 500, EH, 500,
    EH, 500, FH, 350, CH, 150, GS, 500, F, 350, CH, 150, A, 1000,
    AH, 500, A, 350, A, 150, AH, 500, GSH, 250, GH, 250, FSH, 125,
    FH, 125, FSH, 250, ZR, 250, AS, 250, DSH, 500, DH, 250, CSH, 250,
    CH, 125, B, 125, CH, 250, ZR, 250, F, 125, GS, 500, F, 375,
    A, 125, CH, 500, A, 375, CH, 125, EH, 1000, AH, 500, A, 350,
    A, 150, AH, 500, GSH, 250, GH, 250, FSH, 125, FH, 125, FSH, 250,
    ZR, 250, AS, 250, DSH, 500, DH, 250, CSH, 250, CH, 125, B, 125,
    CH, 250, ZR, 250, F, 250, GS, 500, F, 375, CH, 125, A, 500,
    F, 375, CC, 125, A, 1000, ZR, ZR};

void pin_inic(void);
void t2_inic(void);
void tom(int freq, int tempo);

```

```

int main(void) {
    unsigned int i=0;
    unsigned int nota,tempo;
    WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer
    pin_inic();
    t2_inic();
    while(TRUE) {
        nota=marcha[i++];           //Ler nota
        tempo=marcha[i++];          //Ler tempo
        if (nota==ZR && tempo==ZR) i=0; //Repetir?
        tom(freq[nota],tempo);      //Tocar a nota
    }
    return 0;
}

// Gerar sinal na frequência "freq" durante "tp" miliseg
void tom(int freq, int tp){
    int cont;
    if (freq != ZR){             // Freq != zero
        TA2CCR0 = 1048576L/freq; //Programar o período
        TA2CCR2 = TA2CCR0/2;     //Carga de 50%
        cont = freq*(tp/1000.0); //Calcular qtd de coinciências
    }
    else{                        //Se Freq = zero
        TA2CCR0 = 1048576L/1000; //Programar 1 KHz
        TA2CCR2 = 0;             //Sem PWM <<==funciona??
        cont = tp;               //Calcular qtd de coinciências
    }
    while(cont > 0){            //Duração da nota
        while( (TA2CCTL0&CCIFG) == 0); //Esperar TA2CCR0.CCIFG
        TA2CCTL0 &= ~CCIFG;          //Apagar flag
        cont--;                    //Decrementar contador
    }
}

void pin_inic(void){
    P2DIR |= BIT5; //Saída PWM
    P2OUT &= ~BIT5;
    P2SEL |= BIT5;
}

void t2_inic(void){
    TA2CTL = TASSEL_2|ID_0|MC_1|TACLR;
    TA2EX0 = TAIDEX_0;
    TA2CCTL2 = OUTMOD_6;
}

```

Desafio: O leitor é convidado e tentar tocar outras músicas.

Ricardo Zelenovsky e Daniel Café

**ER 7.11.** Provavelmente o leitor já ouviu falar de servo motor, que é um dispositivo eletromecânico cujo movimento (rotação) pode ser comandado. Ele possui um laço de controle em malha fechada e por isso garante a permanência na posição comandada pelo usuário. Na sua forma mais simples, é um motor que permite o controle da posição angular de seu eixo. O mais comum é a informação da posição angular codificada em tensão ou em ciclo de carga de um PWM. No caso, vamos usar o SG90 da Tower Pro que trabalha com PWM, como mostrado na Figura 7.45.

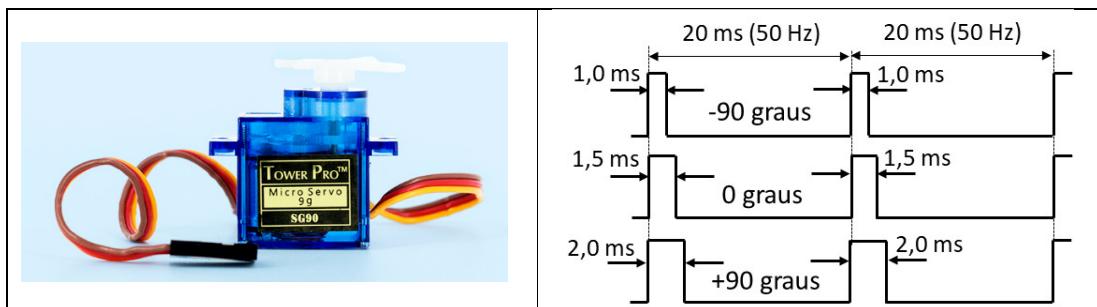


Figura 7.45. Foto do servo motor SG90 e, segundo o manual, a correspondência entre o ângulo e o ciclo de carga do PWM com período de 20 ms.

É pedido um programa que inicie com o eixo do servo em zero graus e use as chaves S1 (P2.1) e S2 (P1.1) para retroceder ou avançar 5 graus, respectivamente. A posição -90 graus deve ser sinalizada com o led vermelho (P1.0) aceso e a posição +90 graus com o led verde (P4.7) aceso.

**Solução:** Antes de iniciar a solução, é preciso informar que o servo motor usado nesta solução, apesar de trazer o rótulo “SG90” não seguiu o que está especificado em seu manual (Figura 7.45). Não sabemos se é uma outra versão do servo ou se é uma contrafação, mas o fato é que ele obedeceu à temporização mostrada na figura abaixo, que foi levantada experimentalmente.

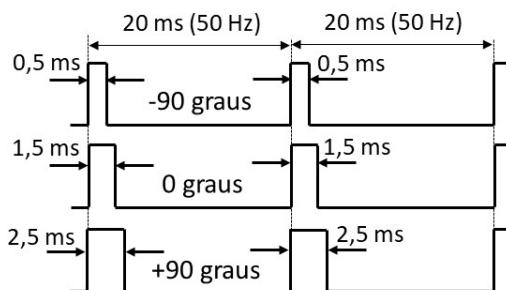


Figura 7.46. Controle do servo motor com os valores medidos na prática e usados no programa solução. Estas medidas conflitam com o manual.

Bem, vamos dar início à solução. Pelos tempos envolvidos, é mais ou menos intuitivo que se deve usar o SMCLK (1.048.576 Hz). O leitor pode tentar fazer as contas abaixo usando o ACLK (32.768 Hz) e se convencer de que a precisão fica muito ruim.

Precisamos programar o contador de TA1 para o modo Ascendente (Up) e limitar a contagem de forma a conseguir 20 ms, que é o período do PWM. Esse limite de contagem é 20.971 ( $1.048.576 \times 0,02$  s) que será carregado no TA1CCR0. O modo de saída será o Modo 6.

Já o pulso dentro deste período varia do mínimo 0,5 ms até o máximo de 2,5 ms. Uma excursão de 2,0 ms, que corresponde a 2.097 contagens. Como essa excursão corresponde à variação de 180 graus, temos que cada 5 graus correspondem a 58 contagens ( $2.097 / (180/5)$ ). A largura deste pulso nunca pode ficar abaixo de 0,5 ms, o que corresponde a 524 contagens ( $1.048.576 \times 0,0005$ ). Para facilitar, o programa definimos as seguintes constantes.

```
#define T20ms    20971 //Período 20 ms (1.048.576 x 0,02)
#define T500us   524    //Período 0,5 ms (1.048.576 x 0,0005) - mínimo
#define P5GRAUS 58     //Passo de 5 graus (2.097 / 36)
```

O programa agora fica simples. Vamos usar a variável `pos` para, sob o comando de S1 e S2, contar entre 0 e 36, que são os 36 passos para varrer a faixa de 180 graus em passos de 5 graus. Se `pos = 0`, acendemos o *led* vermelho e se `pos = 36`, acendemos o *led* verde. O período do PWM é dado pela conta abaixo.

```
TA1CCR1 = T500us + pos*P5GRAUS;           //Programar PWM
```

A listagem a seguir apresenta a solução completa. Nela, as funções `check_s1()`, `check_s2()` e `debounce()` foram omitidas, para evitar que ela ficasse muito extensa. Essas funções devem ser copiadas do ER 7.7.

#### Listagem da solução do ER 7.11

```
// ER 7.11
// Acionar Servo SG90 com TA1.1 (P2.0)
// +5V = fio vermelho
// GND = fio marrom
```

```

// PWM = fio amarelo conectar em P2.0

// Parte com servo em 0 graus (pulso = 1,5 ms)
// S1 aumenta 5 graus, até +90 graus (Led VD aceso)
// S2 diminui 5 graus, até -90 graus (Led VM aceso)

#include <msp430.h>

#define TRUE      1
#define FALSE     0

#define ABT 1      //Constante representa Aberta
#define FEC 0      //Constante representa Fechada
#define DBC 1000   //Atraso para o debounce

#define T20ms    20972 //Período 20 ms (1.048.576 x 0,02)
#define T500us   524    //Período 0,5 ms (1.048.576 x 0,0005)
#define P5GRAUS 58     //Passo de 5 graus (1.048.576 x (0,002/36))

void config_t1(void);           //Config TA1
void config_pinos(void);        //Config Pinos
int check_s1(void);            //Checar chave S1
int check_s2(void);            //Checar chave S2
void debounce(void);            //Delay para debounce

int ps1=ABT,ps2=ABT;           //Estado anterior das chaves
volatile int pos=18;            //Posição em passos de 5 graus (0, 1, ... 36)

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer
    config_t1();
    config_pinos();
    while(TRUE){
        if (check_s2()) pos++; //Avançar 5 graus
        if (check_s1()) pos--; //Recuar 5 graus
        ;
        if (pos > 36) pos=36; //Limite 100%
        if (pos < 0)   pos=0; //Limite 0%
        P4OUT &= ~BIT7;          //VD apagado
        P1OUT &= ~BIT0;          //VM apagado
        if (pos==36)    P4OUT |= BIT7; //VD aceso
        if (pos==0)     P1OUT |= BIT0; //VM aceso
        TA1CCR1=T500us+pos*P5GRAUS; //Programar PWM
    }
    return 0;
}

// TA1.1 com período de 20 ms e PWM = 1,5 ms (0 graus)
void config_t1(void){

```

```

TA1CTL = TASSEL_2 | ID_0 | MC_1 | TACLR;
TA1EX0 = TA1DEX_0;
TA1CCR0 = T20ms; //Período de 20 ms
TA1CCTL1 = OUTMOD_6;
TA1CCR1 = T500us+18*P5GRAUS; //Zero graus
//Configurar P2.0 para PWM
P2DIR |= BIT0;
P2SEL |= BIT0; //P2.0 = PWM
}

int check_s1(void){ ... } //Copiar do ER 7.8
int check_s2(void){ ... } //Copiar do ER 7.8
void config_pinos(void){ ... } //Copiar do ER 7.8
void debounce(void){ ... } //Copiar do ER 7.8

```

**ER 7.12.** Já vimos neste capítulo que timer B é o mais indicado para acionar motor por PWM pois ele possui dupla buferização. O modo de contagem deve ser o Modo 3 (Up/Down), pois garante o melhor posicionamento dos pulsos quando se altera o Ciclo de Carga. Assim, neste exercício vamos usar Timer B0 no Modo 3 (Up/Down) operando com o SMCLK (1.048.576 Hz) e sua unidade de comparação 5 (TB0.5 = P3.5) para acionar um motor com PWM na frequência de 1 kHz. O programa inicia com Ciclo de Carga igual a 50%. As chaves S1 e S2 serão usadas para o controle do ciclo de carga e os dois *leds* para sinalização:

- Chave S1 → incremento de 10% no ciclo de carga;
- Chave S2 → decremento de 10% no ciclo de carga.;
- *Led L1* (vermelho) → aceso toda vez que o ciclo de carga for igual 100% e
- *Led L2* (verde) → aceso toda vez que o ciclo de carga for igual 0%

**Solução:** A primeira pergunta que surge é: como conectar um motor ao MSP430? Nos exercícios passados, quando usamos os *leds*, eles estavam ligados diretamente aos pinos, ou seja, a porta do MSP430 fornecia corrente para o *led*. Isso pôde ser feito porque o consumo do *led* está dentro da capacidade de corrente das portas dos MSP. Quando se trata de um motor, que consome muito mais corrente, isso NUNCA deve ser feito. Temos ainda uma outra preocupação muito maior, pois o motor é uma carga indutiva e gera a força contra eletromotriz quando o campo de sua bobina é cortado.

É fortemente recomendado que o leitor estude o Apêndice M, onde são apresentadas soluções para o acionamento de motores DC com o MSP. A Figura 7.47 apresenta a solução que será usada para este exercício. Ela não é a única. No Apêndice M existe uma série de outras alternativas e o leitor deve escolher, em função do seu motor, a que lhe for mais adequada. Se for um motor de 5 V e baixa corrente (menor que 500 mA), pode ser usada a alimentação de 5 V da Launchpad.

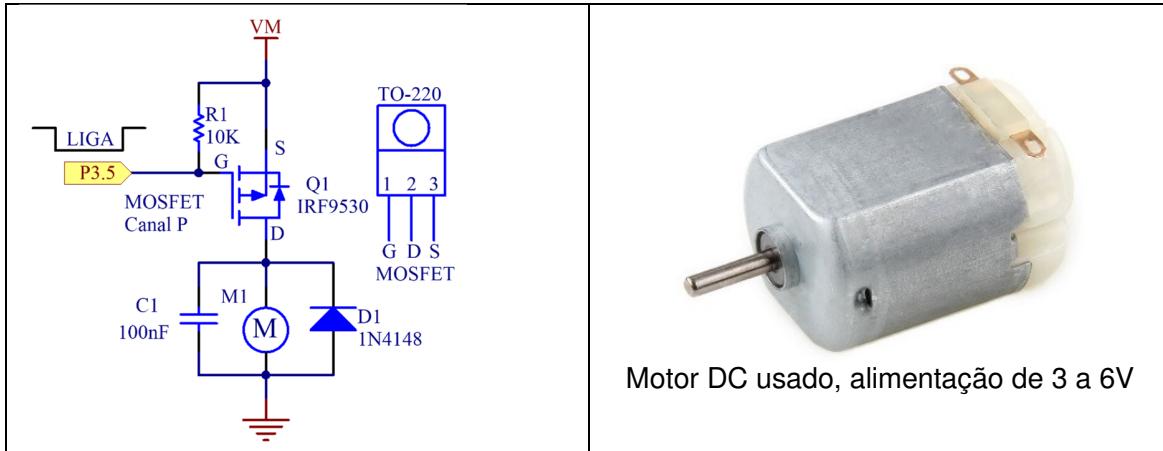


Figura 7.47. Uma solução para acionar um motor DC a partir de um PWM gerado pelo MSP e a foto do motor usado. A tensão VM é a tensão de alimentação do motor que, usualmente, não é a mesma que alimenta o MSP.

Vamos agora tratar da configuração do PWM. Se a frequência é de 1 kHz e estamos usando o SMCLK, então vamos precisar de 1.048 contagens ( $1.048.576 / 1.000$ ) para gerar o período de 1 ms. Lembrando que é o Modo 3, então precisaremos programar a metade (524) no TB0CCR0. Cada passo terá 10% desse valor, o que corresponde a 52,4 contagens. É interessante fazer um ajuste para facilitar a programação. Vamos aproximar o passo de 10% para 52 contagens, assim, 10 passos (100%) corresponderão a 520 contagens ( $TB0CCR0 = 520$ ). Cada acionamento de S1 ou S2 soma ou subtrai 52 (10%) ao Ciclo de Carga.

Apenas mais um detalhe antes de apresentarmos a solução. Note que foi usado um MOSFET de Canal P. O motor é alimentado quando P3.5 vai para nível baixo. Por causa disso, vamos usar a saída no Modo 2 (OUTMODE = 2).

#### Listagem da solução do ER 7.12

```
// ER 7.12
// Acionar motor DC conectado em P3.6 (TB0.6)
// Período do PWM 1000 Hz
// S1 aumenta em 10%, Led vermelho aceso se PWM = 100%
// S2 diminui em 10%, Led verde aceso se PWM = 0%

#include <msp430.h>

#define TRUE 1
#define FALSE 0

#define ABT 1      //Constante representa Aberta
#define FEC 0      //Constante representa Fechada
```

```

#define DBC 1000      //Atraso para o debounce

#define T1ms  520     //Período de 1 ms
#define PASSO 52      //Passo de 10%

void tb0_config(void);          //Configurar TB0.6
void config_pinos(void);       //Config Pinos
int check_s1(void);            //Checar chave S1
int check_s2(void);            //Checar chave S2
void debounce(void);           //Delay para debounce

int ps1=ABT,ps2=ABT;           //Estado anterior das chaves

int main(void)
{
    int pot=0;
    WDTCTL = WDTPW | WDTHOLD;    // stop watchdog timer
    config_pinos();              //Config pinos
    tb0_config();                //Config TB0
    while (TRUE){
        if (check_s1()) pot+=10;
        if (check_s2()) pot-=10;
        if (pot < 0)   pot=0;        //Limite 0%
        if (pot > 100) pot=100;     //Limite 100%
        TB0CCR6 = T1ms-((pot/10)*PASSO); //Recalcular
        P1OUT &= ~BIT0;           //Apagar vermelho
        P4OUT &= ~BIT7;           //Apagar verde
        if (pot==100)  P1OUT |= BIT0; //Acender vermelho
        if (pot==0)    P4OUT |= BIT7; //Acender verde
    }
    return 0;
}

// Configurar TB0.6
void tb0_config(void){
    TB0CTL = TBSSEL_2 | MC_3;      //SMCLK e MC=3
    TB0CCTL0=0;
    TB0CCR0=T1ms;                //Período de 1 ms (Modo 3)
    TB0CCTL6= CLLD_2 | OUTMOD_2;  //P3.6 = PWM acionar motor
    TB0CCR6=TB0CCR0;             //Iniciar com Ciclo de Carga 0%
    //Configurar pino P3.6 para PWM
    P3DIR |= BIT6;               //TB0.6 = P3.6
    P3SEL |= BIT6;               //Usar com PWM
}

int check_s1(void){ ... }      //Copiar do ER 7.7
int check_s2(void){ ... }      //Copiar do ER 7.7
void config_pinos(void){ ... } //Copiar do ER 7.7
void debounce(void){ ... }     //Copiar do ER 7.7

```

O programa é relativamente simples e dispensa maiores explicações, com exceção da linha abaixo.

```
TB0CCR6 = T1ms - ((pot/10) *PASSO); //Recalcular
```

O contador `pot` vai de 0 até 100, entretanto, preparamos para 10 passos. Assim, ele precisa ser dividido por 10 antes de ser multiplicado pela constante `PASSO`. Um exame na Figura 7.48 vai mostrar que:

- se  $TB0CCR6 = TB0CCR0 \rightarrow$  potência = 0%
- se  $TB0CCR6 = 0 \rightarrow$  potência = 100%

Por isso, precisamos subtrair a contagem calculada ( $((pot/10) *PASSO)$  do período total (`T1ms`), antes de carregar em `TB0CCR6`.

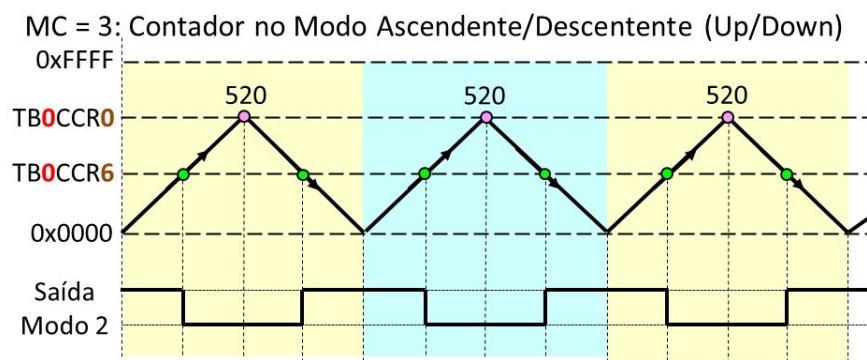


Figura 7.48. Geração da saída no Modo 2 (`OUTMODE = 2`) com o contador `TB0` contando no modo Ascendente/descendente ( $MC = 3$ ).

**ER 7.13.** Neste exercício, vamos estudar o recurso de captura do timer e, para facilitar, vamos trabalhar num ambiente bem simples. Use o TA1.1 (P2.0), acionado pelo ACLK, para gerar uma onda quadrada de 1 kHz e use o recurso de captura de TA2.1 (P2.4), acionado pelo SMCLK, para conferir a frequência do sinal. O led verde (P4.7) acende se o período estiver dentro do esperado, caso contrário, acende o led vermelho (P1.0). É claro, será preciso conectar P2.0 com P2.4, como mostrado na figura a seguir.

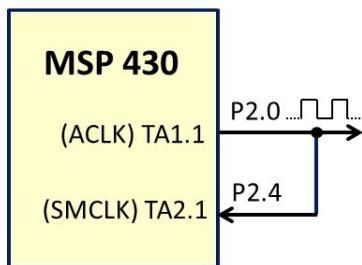


Figura 7.49. Esquema para usar o timer TA2.1 (P2.4) para conferir a onda quadrada pelo TA1.1 em P2.0.

**Solução:** A primeira parte desta solução consiste em programar TA1.1 (P2.0) para gerar uma onda quadrada de 1 kHz, usando o ACLK (32.768). Com o ACLK, em 1 s temos 32.768 contagens, então em 1 ms teremos 32,7 contagens. Aproximamos para 33. Como o zero é também contado, vamos usar 32 contagens. Então programamos:

- TA1CCR0 = 32 e TA1CCR1 = 16 (50% do período).

Para a segunda parte, vamos usar a captura por flanco de subida de TA2.1 que opera com SMCLK (1.048.576) para conferir o período desta onda quadrada. É fácil de ver que o valor esperado é 1.048. A listagem a seguir apresenta o programa solução. Ao executá-la, o leitor vai constatar que o led vermelho fica sempre aceso, ou seja, parece que o período está errado. O leitor consegue explicar esse fato?

#### Listagem para a solução “a” do ER 7.13

```
// ER 7.13.a

// (TA1.1, ACLK) P2.0 gera onda quadrada de 1 kHz
// (TA2.1, SMCLK) P2.4 usa captura para conferir o período
// Colocar em curto P2.0 com P2.4

// Período correto --> ler verde (P4.7)
// Período errado --> ler vermelho (P1.0)

#include <msp430.h>

#define TRUE      1
#define T1ms     32      //Período 1ms para TA1 (ACLK)
#define CAP_OK   1048    //Período 1ms medido por TA2 (SMCLK)

void ta1_config(void); //Configurar TA1.1
void ta2_config(void); //Configurar TA2.1
void leds_config(void); //Configurar leds

int main(void)
```

```

{
    unsigned int velho, novo;      //Captura antiga e atual
    int dif;                      //Diferença nas capturas
    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer
    leds_config();
    ta1_config();
    ta2_config();
    velho=novo=0;
    while(TRUE){                  //Laço infinito
        while( (TA2CCTL1&CCIFG) == 0); //Captura?
        TA2CCTL1 &= ~CCIFG;           //Apagar flag
        novo=TA2CCR1;                //Ler captura
        dif = novo-velho;             //Calcular diferença
        velho=novo;                  //Novo vira velho
        if (dif<0) dif += 65536L;     //Ver diferença negativa
        P4OUT &= ~BIT7;               //VD apagado
        P1OUT &= ~BIT0;               //VM apagado
        if (dif==CAP_OK) P4OUT |= BIT7; //OK? => VD aceso
        else                 P1OUT |= BIT0; //NOK? => VM aceso
    }
    return 0;
}

// Onda quadrada com TA1.1 (P2.0)
void ta1_config(void){
    TA1CTL=TASSEL_1 | MC_1; //ACLK e Modo 1
    TA1CCR0 = T1ms;          //Período de 1 ms
    TA1CCTL1 = OUTMOD_6;     //Modo de saída
    TA1CCR1 = TA1CCR0/2;     //50% de ciclo de carga
    P2DIR |= BIT0;            //Habilitar saída
    P2SEL |= BIT0;            //por P2.0
}

// Captura com TA2.1 (P2.4)
void ta2_config(void){
    TA2CTL=TASSEL_2 | MC_2; //SMCLK e Modo 2 (continuo)
    TA2CCTL1 = CM_1 | CCIS_0 | SCS | CAP; //Captura flanco subida
                                            //Selecionar entrada CCI2A (P2.4)
                                            //Captura síncrona
                                            //Habilitar Modo Captura
    P2DIR &= ~BIT4;                //P2.4 = entrada
    P2SEL |= BIT4;                 //dedicada à captura
}
void config_leds(void){...} //Copiar do ER 7.1

```

Uma primeira resposta para o fato do programa acima sempre indicar erro tem relação com a frequência dos relógios ACLK e SMCLK. Usamos o ACLK (32.768 Hz) para gera a

onda quadrada e fizemos a conferência usando o SMCLK (1.048.576 Hz). A resolução do SMCLK é maior que a do ACLK, como mostrado abaixo.

- $T_{ACLK} = 1/32.768 = 30,518 \mu s$
- $T_{SMCLK} = 1/1.046.576 = 0,954 \mu s$

Então, é de se esperar algum erro no sinal gerado. A relação entre SMCLK e ACLK é de 32. Então, poderíamos esperar um erro de até 32 contagens. Fazemos essa afirmação sem levar em conta se esses relógios estão ou não sincronizados. Ver Capítulo sobre a UCS para esclarecer essa dúvida. Para efeito deste exercício, vamos estabelecer uma tolerância de 32 contagens. Assim, o período medido e o esperado podem diferir em até 32 contagens. A listagem a seguir (um pouco resumida) apresente uma alteração que leva em conta essa tolerância. O leitor é convidado a verificar qual o limite dessa tolerância (`#define TOL 32`) e também a inserir um erro no período da onda quadrada (`#define T1ms 32`).

### Listagem para a solução “b” do ER 7.13

```
// ER 7.13.b - inclui tolerância no cálculo do erro

// (TA1.1, ACLK) P2.0 gera onda quadrada de 1 kHz
// (TA2.1, SMCLK) P2.4 usa captura para conferir o período
// Colocar em curto P2.0 com P2.4

// Período correto --> ler verde (P4.7)
// Período errado --> ler vermelho (P1.0)

#include <msp430.h>

#define TRUE      1
#define T1ms      32      //Período 1ms para TA1 (ACLK)
#define CAP_OK    1048    //Período 1ms medido por TA2 (SMCLK)
#define TOL       32      //Tolerância de erro no cálculo do período

void ta1_config(void); //Configurar TA1.1
void ta2_config(void); //Configurar TA2.1
void leds_config(void); //Configurar leds

int main(void)
{
    unsigned int velho, novo;    //Captura antiga e atual
    unsigned int erro;           //Erro em esperado e calculado
    int dif;                    //Diferença nas capturas
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer
    leds_config();
    ta1_config();
    ta2_config();
    velho=novo=0;
}
```

```

while(TRUE) {                                //Laço infinito
    while( (TA2CCTL1&CCIFG) == 0);          //Captura?
    TA2CCTL1 &= ~CCIFG;                      //Apagar flag
    novo=TA2CCR1;                           //Ler captura
    dif = novo-velho;                      //Calcular diferença
    velho=novo;                            //Novo vira velho
    if (dif<0)   dif += 65536L;             //Ver diferença negativa
    erro=abs(CAP_OK-dif);                  //Calcular erro
    P4OUT &= ~BIT7;                         //VD apagado
    P1OUT &= ~BIT0;                         //VM apagado
    if (erro<TOL)   P4OUT |= BIT7;           //OK? => VD aceso
    else           P1OUT |= BIT0;           //NOK? => VM aceso
}
return 0;
}
void ta1_config(void){ ... } //Copiar da versão anterior
void ta2_config(void){ ... } //Copiar da versão anterior
void leds_config(void){ ... } //Copiar da versão anterior

```

Para finalizar, vamos resolver o mesmo exercício, mas agora usando a interrupção da captura. Nesta solução, note que as variáveis são globais e estão marcadas como `volatile`. A variável `flag` foi introduzida para facilitar a coordenação entre o programa principal e a rotina de interrupção. A interrupção, calcula o erro e faz `flag = TRUE`. O programa principal fica esperando por `flag = TRUE`. Quando isso acontece, ele faz `flag = FALSE`, para a próxima repetição, e atualiza os `leds` em função do `erro`. Como habilitamos uma única interrupção, não foi necessário consultar o valor de `TA2IV`, apenas fazemos uma leitura (`TA2IV`;) para zerar a flag `CCIFG`.

O leitor é convidado a alterar esse programa de forma que o controle dos `leds` seja feito dentro da subrotina de interrupção. Lembramos que a rotina que atende à interrupção deve ser a mais curta possível. Assim, a melhor solução é usar a interrupção apenas para a captura e ativar a flag. Tudo o resto deveria ser feito pelo programa principal.

#### Listagem para a solução “c” do ER 7.13

```

// ER 7.13.c - inclui tolerância e interrupção

#include <msp430.h>

#define TRUE      1
#define FALSE     0
#define T1ms      32      //Período 1ms para TA1 (ACLK)
#define CAP_OK    1048    //Período 1ms medido por TA2 (SMCLK)
#define TOL       32      //Tolerância de erro no cálculo do período

```

```

void ta1_config(void); //Configurar TA1.1
void ta2_config(void); //Configurar TA2.1
void leds_config(void); //Configurar leds

volatile unsigned int flag; //Coordenar com interrupção
volatile unsigned int velho, novo,erro; //Capturas e erro
volatile int dif; //Diferença nas capturas

int main(void){
    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer
    leds_config();
    ta1_config();
    ta2_config();
    velho=novo=0;
    __enable_interrupt();
    while(TRUE){ //Laço infinito
        while(flag==FALSE); //Esperar Captura
        flag=FALSE; //Prep flag para a próxima
        P4OUT &= ~BIT7; //VD apagado
        P1OUT &= ~BIT0; //VM apagado
        if (erro<TOL) P4OUT |= BIT7; //OK? => VD aceso
        else P1OUT |= BIT0; //NOK? => VM aceso
    }
    return 0;
}

// Interrupção Captura TA2.1
// #pragma vector = TIMER2_A1_VECTOR
#pragma vector = 43
__interrupt void ta2_cap(void){
    TA2IV; //Ler TA2IV para zerar CCIFG
    novo=TA2CCR1; //Ler captura
    dif = novo-velho; //Calcular diferença
    velho=novo; //Novo vira velho
    if (dif<0) dif += 65536L; //Ver diferença negativa
    erro=abs(CAP_OK-dif); //Calcular erro
    flag=TRUE;
}

// Captura com TA2.1 (P2.4)
void ta2_config(void){
    TA2CTL=TASSEL_2 | MC_2; //SMCLK e Modo 2 (contínuo)
    TA2CCTL1 = CM_1 | CCIS_0 | SCS | CAP | CCIE; //Hab. interrupção
    P2DIR &= ~BIT4; //P2.4 = entrada
    P2SEL |= BIT4; //dedicada à captura
}

void ta1_config(void){ ... } //Copiar da versão anterior
void leds_config(void){ ... } //Copiar da versão anterior

```

**ER 7.14.** Neste exercício vamos usar o TA0.0. A entrada da unidade de captura 0 deste *timer* está ligada ao pino P1.1, onde também está conectada a chave S2. Vamos então fazer algo não muito usual: construir um contador de acionamentos de S2 usando o modo de captura. Já podemos esperar que os rebotes causem problemas. Vamos usar os dois *leds* para construir um contador binário de 2 bits, como mostrado abaixo, e assim poderemos observar se ocorrem rebotes.

Tabela 7.17. Contador binário usando os dois *leds* da LaunchPad

Contador 2 bits	Led Vermelho (P1.0)	Led Verde (P4.7)
0	0	0
1	0	1
2	1	0
3	1	1

0 = *led* apagado e 1 = *led* aceso

**Solução:** O exercício é bem simples. Programamos a unidade de captura 0 para trabalhar com flancos de descida. Depois, ficamos em um *polling* esperando CCIFG = 1, que caracteriza a captura de um flanco. A cada flanco, incrementamos o contador *cont* e acionamos os *leds*. Merece comentário a linha marcada em cinza, que zera o flag CCIFG antes de entrar no laço infinito. O leitor pode se perguntar por que fazer isso, se o programa ainda nem começou. Esta é uma preocupação importante, pois a configuração do pino de captura ou do *timer* pode gerar um falso flanco positivo. O leitor é convidado a testar o programa, comentando essa linha.

Uma maneira simples de construir um contador com os dois *leds* foi usando os dois *bits* menos significativos do contador *ct*. A função void *trata\_leds*(unsigned int *ct*) apaga os dois *leds* e depois verifica qual deve ficar aceso. Isto facilita a escrituração da função e é tão rápido que o usuário não percebe.

Listagem para a solução “a” do ER 7.14

```
// ER 7.14.a
// Contar acionamentos de S2 usando TA0.0 (P1.1)

#include <msp430.h>

#define TRUE 1

void ta0_config(void);
```

```

void trata_leds(unsigned int ct);
void config_leds(void);

int main(void)
{
    volatile unsigned int cont=0;
    WDTCTL = WDTPW | WDTHOLD;           // stop watchdog timer
    leds_config();
    ta0_config();
    trata_leds(cont);
    TA0CCTL0 &= ~CCIFG; //Apagar possível CCIFG=1
    while(TRUE){                      //Laço infinito
        while ( (TA0CCTL0&CCIFG)==0); //Captura?
        TA0CCTL0 &= ~CCIFG;          //Apagar CCIFG
        cont++;                     //Incrementar contador
        trata_leds(cont);          //Acender leds
    }
    return 0;
}

// Acender leds de acordo com o contador
void trata_leds(unsigned int ct){
    P1OUT &= ~BIT0;      //Apaga vermelho
    P4OUT &= ~BIT7;      //Apaga verde
    switch(ct&3){        //Testar 2 bits do contador
    case 0: break;
    case 1: P4OUT |= BIT7; break;
    case 2: P1OUT |= BIT0; break;
    case 3: P4OUT |= BIT7; P1OUT |= BIT0; break;
    }
}

// Captura TA0.0 (P1.1 = S1)
void ta0_config(void){
    TA0CTL=TASSEL_1 | MC_2; //ACLK e Modo 2
    TA0CCTL0 = CM_2 |       //Captura flanco descida
               CCIS_0 |     //Selecionar entrada CCI0A (P1.1)
               SCS |         //Captura síncrona
               CAP;          //Habilitar Modo Captura
    P1DIR &= ~BIT1;        //P1.1 = entrada
    P1REN |= BIT1;         //Habilitar resistor
    P1OUT |= BIT1;         //Selecionar Pullup
    P1SEL |= BIT1;         //Captura
}
void config_leds(void){...} //Copiar do ER 7.1

```

Vamos agora resolver este mesmo exercício, mas usando a interrupção da captura. A solução é muito simples. Note que agora o programa principal fica preso num laço infinito vazio, já que tudo é feito pela interrupção.

#### Listagem para a solução “b” do ER 7.14

```
// ER 7.14.b
// Contar acionamentos de S2 usando TA0.0 (P1.1)
// Agora com interrupção de captura

#include <msp430.h>

#define TRUE 1

void ta0_config(void);
void trata_leds(unsigned int ct);
void leds_config(void);

volatile unsigned int cont=0;

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;      // stop watchdog timer
    leds_config();
    ta0_config();
    trata_leds(cont);
    __enable_interrupt();
    TA0CCTL0 &= ~CCIFG; //Apagar possível CCIFG=1
    while(TRUE);           //Laço infinito
    return 0;
}

// Interrupção Captura TA0.0
// #pragma vector = TIMER0_A0_VECTOR
#pragma vector = 53
__interrupt void ta0_cap(void){
    cont++;                  //Incrementar contador
    trata_leds(cont);       //Acender leds
}

void ta0_config(void){ ... }           //Copiar do exercício anterior
void trata_leds(unsigned int ct){ ... } //Copiar do exercício anterior
void leds_config(void){ ... }         //Igual à config_leds(void)
```

**ER 7.15.** Já vimos que o acionamento de uma chave com contatos metálicos provoca o surgimento de rebotes, como ilustrado na Figura 7.50. Neste exercício vamos usar a captura de TA0.0 para medir o intervalo entre os rebotes de S2 (P1.1). Os possíveis dez primeiros rebotes devidos ao acionamento deverão ser armazenados no vetor `unsigned`

`int vaf[10]` e os possíveis 10 primeiros rebotes devidos à liberação da chave deverão ser armazenados no vetor `unsigned int vfa[10]`. A quantidade máxima de rebotes foi arbitrada em 10, o que é discutível, porém a prática se mostrou adequado.

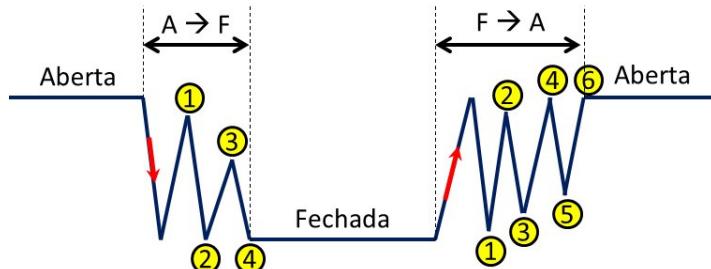


Figura 7.50. Ilustração dos rebotes ao acionar e liberar uma chave.

Para facilitar a execução e a sincronização deste exercício, vamos usar S1 e os *leds* para definir os seguintes estados:

- *Led verde aceso* → esperando acionar S2;
- *Led vermelho aceso* → esperando soltar chave S2 e
- Nenhum *led* aceso → usar pausa do CCS para examinar os resultados (vetores `vaf` e `vfa`) e depois acionar S1 para repetir o exercício

**Solução:** Programamos a unidade de captura 0 para capturar ambos flancos (MC\_3) e habilitamos sua interrupção. Como existe só um vetor de interrupção para esta unidade (TA0CCTL0.CCIFG), a rotina fica bem simples. O laço infinito está esquematizado a seguir e, na partida, os vetores estão zerados.

- 1) Acender *led* verde, esperar acionar S2;
- 2) Com S2 acionada, esperar algum tempo para interrupção capturar flancos;
- 3) Apagar *led* verde e acender *led* vermelho, esperar liberar S2;
- 4) Com S2 liberada, esperar algum tempo para interrupção capturar flancos;
- 5) Apagar *led* vermelho e esperar usuário acionar S1. Essa é a hora de pausar o CCS e verificar os resultados nos vetores `vaf` e `vfa`;
- 6) Com S1 acionada, zerar os vetores e voltar para passo 1.

É importante ressaltar que é preciso habilitar o *pullup* de S2 (P1.1), o que é feito na função `ta0_config()`. A execução desta solução vai mostrar que, na maioria das vezes, se captura um ou dois flancos. A primeira captura corresponde ao acionamento da chave, as demais são os rebotes. A diferença entre valores consecutivos dos vetores corresponde, em contagens de SMCLK (1.048.576 Hz), ao intervalo entre os dois eventos. O leitor é convidado a tentar acionar S2 de diferentes formas para ver o que acontece com os rebotes.

Pergunta: Por que, algumas vezes, posições consecutivas do vetor têm o mesmo valor?

### Listagem solução do ER 7.15

```
// ER 7.15
// TA0.0 (P1.1) medir instante dos rebotes de S2

#include <msp430.h>

#define TRUE      1
#define FALSE     0
#define CONTAR   10000    //Período de espera por flancos
#define MAX       10      //Máximo de rebotes

void ta0_config(void);
void leds_s1_config(void);

volatile unsigned int vaf[MAX]; //Flancos A-->F
volatile unsigned int vfa[MAX]; //Flancos F-->A
volatile unsigned int af,ix;    //Definir flanco e indexar dos vetores

int main(void){
    volatile int cont;
    WDTCTL = WDTPW | WDTHOLD;    // stop watchdog timer
    leds_s1_config();
    ta0_config();
    for (ix=0; ix<MAX; ix++){  //Zerar vetores
        vaf[ix]=0;
        vfa[ix]=0;
    }
    __enable_interrupt();
    while(TRUE){                //Laço infinito
        P4OUT |= BIT7;          //Verde aceso
        af=TRUE;                //Marcar A-->F
        ix=0;                   //Zerar indexador
        while ( (P1IN & BIT1) == BIT1); //Esperar acionar
        for (cont=0; cont<CONTAR; cont++); //Aguardar rebotes
        P4OUT &= ~BIT7;           //Verde apagado
        P1OUT |= BIT0;            //Vermelho aceso
        af=FALSE;                //Marcar F-->A
        ix=0;                   //Zerar indexador
        while ( (P1IN & BIT1) == 0); //Esperar soltar
        for (cont=0; cont<CONTAR; cont++); //Aguardar rebotes

        P1OUT &= ~BIT0;           //Vermelho apagado
        while ( (P2IN & BIT1) == BIT1); //Esperar S1 para repetir
        for (ix=0; ix<MAX; ix++){ //Zerar vetores
            vaf[ix]=0;
            vfa[ix]=0;
        }
    }
}
```

```

        }
    }
    return 0;
}

// Interrupção Captura TA0.0
// #pragma vector = TIMER0_A0_VECTOR
#pragma vector = 53
__interrupt void ta0_cap(void) {
    if (ix<10){
        if (af) vaf[ix++]=TA0CCR0; //A-->F
        else     vfa[ix++]=TA0CCR0; //F-->A
    }
}

// Captura TA0.0 (P1.1 = S1)
void ta0_config(void){
    TA0CTL=TASSEL_2 | MC_2; //SMCLK e Modo 2
    TA0CCTL0 = CM_3 | CCIS_0 | SCS | CAP | CCIE; //Hab. Interrup. também
    P1DIR &= ~BIT1;           //P1.1 = entrada
    P1REN |= BIT1;            //Habilitar resistor
    P1OUT |= BIT1;            //Selecionar Pullup
    P1SEL |= BIT1;            //Captura
}

// Configurar leds e chave S1
void leds_s1_config(void){
    P1DIR |= BIT0; P1OUT &= ~BIT0; //Led Vermelho
    P4DIR |= BIT7; P4OUT &= ~BIT7; //Led Verde
    P2DIR &= ~BIT1;           //P2.1 = entrada
    P2REN |= BIT1;            //Habilitar resistor
    P2OUT |= BIT1;            //Selecionar Pullup
    P2SEL |= BIT1;            //Captura
}

```

**ER 7.16.** No ER 7.12 preparamos um ambiente simples para controlar a velocidade de rotação de um motor DC. Agora vamos propor um aperfeiçoamento, que exigirá um pouco de habilidade mecânica do leitor. A sugestão é a de prender no eixo do motor um pedaço de canudo plástico, desses que estão proibidos, de forma a projetar sombra sobre um sensor de luz infravermelha, preso ao corpo do motor. Cada vez que o sensor for sombreado, é porque o canudo está passando sobre ele. O intervalo entre dois sombreamentos consecutivos corresponde ao tempo de meia volta. O anexo M apresenta este mesmo exercício, mas de uma forma muito mais sofisticada. A Figura 7.51 dá uma boa ideia da montagem.

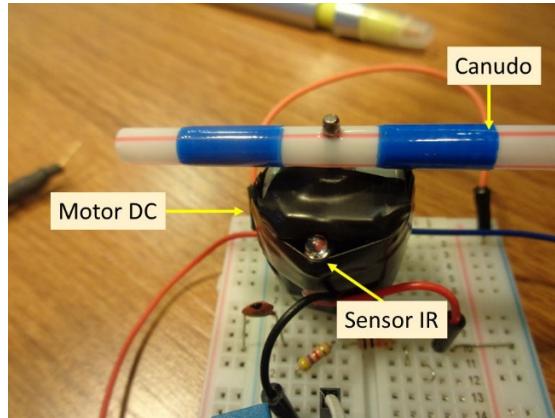


Figura 7.51. Ilustração do motor DC com um canudo preso ao seu eixo e do sensor de luz infravermelho fixado no corpo do motor, logo abaixo do plano de rotação do canudo.

A Figura 7.52 apresenta um diagrama de blocos onde fica bem claro os elementos participantes deste exercício e o circuito elétrico sugerido. O sensor infravermelho (IR) é o TIL78. Ele é um fototransistor. Quando há suficiente incidência de luz, ele satura (vira um curto circuito). Quando sombreado, ele entra em corte (circuito aberto). Assim, cada vez que o canudo fizer sombra sobre o sensor, sua saída vai para nível alto. O potenciômetro R3 ajuda regular a sensibilidade. Quanto maior R3, mais fácil é conseguida a saturação. Para que o sensor funcione bem, é preciso iluminá-lo com uma lanterna, mas o melhor é desempenho é conseguido com uma luminária de mesa.

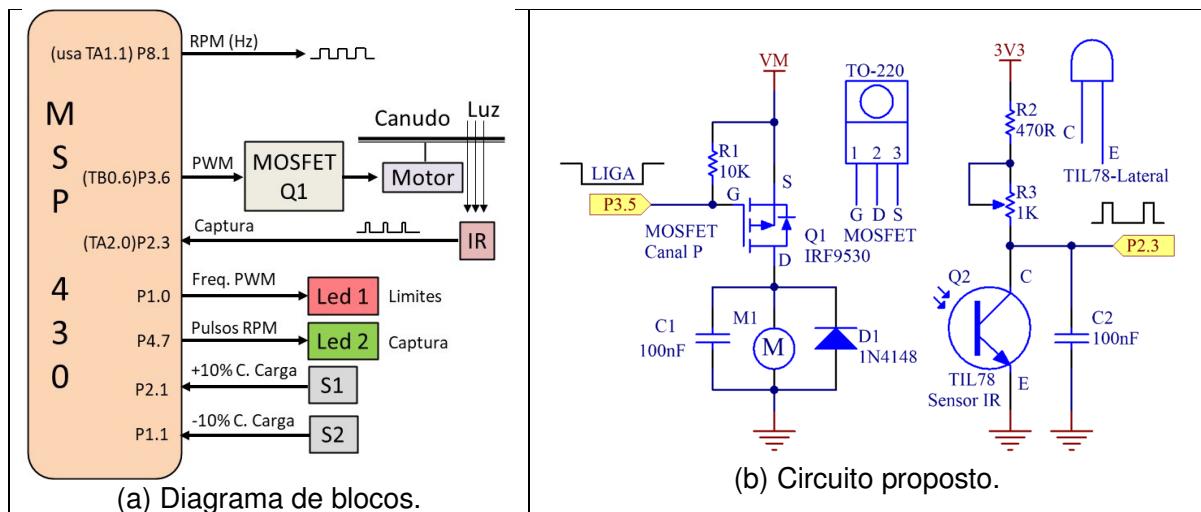


Figura 7.51. Diagrama de blocos da solução sugerida e uma proposta para o circuito elétrico.

As chaves continuam desempenhando o mesmo papel que desempenhavam no ER 7.12:

- Chave S1 → incremento de 10% no ciclo de carga;
- Chave S2 → decremento de 10% no ciclo de carga.;

Porém, vamos alterar um pouco a sinalização feita pelos *leds*.

- *Led vermelho* → só acende com ciclo de carga igual a 0% ou 100% e
- *Led verde* → inverte de estado a cada captura.

Assim, com *led vermelho* temos como acompanhar o ciclo de carga. O *led verde* ajuda na regulagem do infravermelho. Observando o *led verde* e girando o eixo do motor com a mão é possível ver se a captura está funcionando. Esse *led* inverte de estado cada vez que for sombreado.

Para finalizar, é preciso uma forma simples de indicar a velocidade de rotação do motor em RPM (rotações por minuto). Hoje em dia, é fácil conseguir um multímetro que ofereça a opção de frequencímetro. Então, pelo pino P8.1 vamos gerar uma onda quadrada com frequência numericamente idêntica à velocidade em RPM. Assim, basta ligar o frequencímetro em P8.1 para saber a velocidade de rotação.

**Solução:** A solução é um pouco longa, mas se aproveita tudo o que foi feito no ER 7.12, só precisaremos alterar o funcionamento dos *leds*. A esta solução é preciso adicionar a captura dos pulsos IR, o cálculo da velocidade em RPM e a sinalização por P8.1.

Para usar a captura e calcular a velocidade em RPM, temos que decidir qual relógio usar. A tabela abaixo apresenta os limites quando se usa cada um dos relógios disponíveis para medir a rotação. Por exemplo, para o ACLK, o maior intervalo que se consegue medir é de 1 s. Assim, se eixo der meia volta a cada segundo, temos 30 RPM. Por outro lado, o menor intervalo que se consegue medir é de 1/32.768, que corresponde a 30,518 µs. Se o eixo der meia volta nesse período, teremos a velocidade de 983.040 RPM ( $60 \times 16.384$ ). Assim, o ACLK é o mais adequado e nem vamos precisar usar os divisores que o *timer* nos oferece.

Tabela 7.18. Limites de medição de velocidade de giro para cada relógio

Relógios	Freq. Hz	Mínimo Giro	Máximo Giro
SMCLK	1.048.576	480 RPM	$31,4 \times 10^6$ RPM
ACLK	32.768	30 RPM	983.040 RPM

Vamos então programar para que o TA2.0 faça uma captura a cada flanco de subida. Usando a interrupção correspondente, preenchemos um vetor com as contagens realizadas entre as capturas, ou seja, um vetor com os intervalos de tempo gastos para o eixo do motor dar meia volta. Usando a variável *flag*, alertamos ao programa principal que calcula a média e depois a velocidade em RPM usando a equação abaixo (no programa

ela é feita com a precisão float). Usamos o recurso da média para evitar flutuações excessivas nesta medida.

$$RPM = 60 \times \frac{ACLK}{2 \times media}$$

Outro ponto que merece comentário é a geração da onda quadrada numa frequência numericamente igual à velocidade em RPM. Vamos usar o TA1.1 e a sugestão apresentada na página 465 do manual “*MSP 430 – User Guide*”, (item 17.2.3.3 *Use of Continuous Mode*), que foi estudada no ER 7.4. Basicamente, com o contador operando no Modo Contínuo, a cada interrupção de TA1CCTL1.CCIFG invertemos o estado de P8.1 e incrementamos TA1CCR1 para nos interromper a uma certa quantidade de contagens adiante. Essa quantidade de contagens está armazenada na variável `rpm_passo`. Veja na listagem o trecho marcado em cinza.

#### Listagem solução do ER 7.16

```
// ER 7.16
// Acionar motor DC conectado em P3.6 (TB0.6)
// Período do PWM 1000 Hz
// S1 aumenta em 10%,
// S2 diminui em 10%,
// Led vermelho aceso se PWM = 0% ou 100%
// Led verde inverte a cada captura
// (TA2.0) P2.3 faz captura dos flancos de subida
// (TA1.1) P8.1 gera freq numericamente igual à vel. RPM

#include <msp430.h>

#define TRUE 1
#define FALSE 0

#define ABT 1          //Constante representa Aberta
#define FEC 0          //Constante representa Fechada
#define DBC 1000       //Atraso para o debounce

#define RPM_MIN      50    //Mínimo valor de RPM
#define RPM_MAX     10000   //Máximo valor de RPM

#define SMCLK 1048576L //Freq do SMCLK
#define ACLK 32768      //Freq do ACLK
#define T1ms 520         //Período de 1 ms (1 kHz)
#define PASSO 52          //Passo de 10% no ciclo de carga
#define QTD 8             //Qtd de medidas para a média do RPM

void tal_config(void);      //TA1.1 gerar RPM em Hz
```

```

void ta2_config(void);           //TA2.0 Captura pulsos IR
void tb0_config(void);          //TB0.6 gerar PWM
void config_pinos(void);        //Config Pinos
int check_s1(void);             //Checar chave S1
int check_s2(void);             //Checar chave S2
void debounce(void);            //Delay para debounce

volatile unsigned int vet[QTD],ix; //Receber diferenças captura
volatile unsigned int flag;       //Indicar QTD capturas
volatile unsigned int novo,velho; //Capturas instantâneas
volatile unsigned int rpm_passo; //Passo para TA2CCR1 gerar RPM Hz
volatile unsigned int rpm;       //RPM instantânea

int ps1=ABT,ps2=ABT;           //Estado anterior das chaves

int main(void){
    int pot=0,i;               //Potência de 0 até 100%
    unsigned long soma;         //Somatório para calcular a média
    unsigned int media;         //média
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer
    config_pinos();             //Config pinos
    tb0_config();               //Config TB0 ==> PWM
    ta2_config();               //Config TB0 ==> Captura
    tal_config();               //Config TB0 ==> RPM em Hz
    flag=FALSE;                 //Flag para indicar QTD capturas
    rpm_passo=SMCLK/(2*RPM_MIN); //Iniciar mostrando Min RPM
    __enable_interrupt();

    while (TRUE){
        if (check_s1()) pot+=10;
        if (check_s2()) pot-=10;
        if (pot < 0) pot=0;           //Limite 0%
        if (pot > 100) pot=100;        //Limite 100%
        TB0CCR6 = T1ms-((pot/10)*PASSO); //Recalcular
        P1OUT &= ~BIT0;              //Apagar vermelho
        if ( pot==100 || pot==0) P1OUT |= BIT0; //0% ou 100% -> vermelho
        if (flag==TRUE){             //QTD capturas ?
            flag=FALSE;              //Para o próximo laço
            soma=0;
            for (i=0; i<QTD; i++) soma+=vet[i];
            media=soma/QTD;           //Calcular a média
            rpm=((float)ACLK/(2*media))*60; //Calcular RPM
            if (rpm<RPM_MIN) rpm_passo=SMCLK/(2*RPM_MIN); //Passo
            else if (rpm>RPM_MAX) rpm_passo=SMCLK/(2*RPM_MAX); //Passo
            else rpm_passo=SMCLK/(2*rpm); //Passo
        }
    }
    return 0;
}

```

```

// TA2.0 Interrupt - Captura e cálculo RPM
//#pragma vector = TIMER2_A0_VECTOR
#pragma vector = 44
__interrupt void ta2_ccr0(void) {
    long d;
    novo=TA2CCR0; //nova captura
    d=(unsigned long)novo-(unsigned long)velho;
    if (d<0) d+=65536L;
    vet[ix++]=d; //Armazenar no vetor
    if (ix == QTD){ //Verificar limite
        flag=TRUE; //Indicar que tem QTD medidas
        ix=0; //Zerar indexador
    }
    P4OUT ^= BIT7; //Inverter led verde
    velho=novo; //Para a próxima captura
}

// TA1.1 Interrupt - Gerar RPM em Hz
//#pragma vector = TIMER1_A1_VECTOR
#pragma vector = 48
__interrupt void ta1_ccr1(void){
    TA1IV; //Zerar CCIFG
    P8OUT^=BIT1; //TA1.1 = RPM = P8.1
    TA1CCR1+=rpm_passo; //Prep para prox interrup
}

// TA1.1 --> P8.1 = RPM Hz = onda quadrada
void ta1_config(void){
    TA1CTL = TASSEL_2 | MC_2; //SMCLK, MC=Continuo
    TA1CCTL1 = CCIE; //Hab int CCR1 (RPM)
    TA1CCR1 = 0; //para o RPM em Hz
    P8DIR |= BIT1; //P8.1=Saida
    P8OUT &= ~BIT1; //P8.1=0
}

// TA2.0 --> (P2.3) Captura para calcular RPM
void ta2_config(void){
    TA2CTL = TASSEL_1 | MC_2; //ACLK, MC=Continuo
    TA2CCTL0 = CM_1 | SCS | CAP | CCIE; //Captura pulsos IR
    P2DIR &= ~BIT3; //P2.3 recebe pulsos IR
    P2REN |= BIT3; //Resistor
    P2OUT |= BIT3; //Pullup
    P2SEL |= BIT3; //Dedicar à captura
}

void tb0_config(void){...} //Copiar do ER 7.12
int check_s1(void){...} //Copiar do ER 7.12
int check_s2(void){...} //Copiar do ER 7.12
void config_pinos(void){...} //Copiar do ER 7.12

```

```
void debounce(void) {...} //Copiar do ER 7.12
```

**ER 7.17.** Neste exercício e no próximo vamos voltar a usar PWM. O processador MSP430 F5529 não tem um Conversor Digital-Analógico (DAC). Entretanto, tal dispositivo pode ser construído de forma simples usando um gerador PWM e um filtro passa-baixa. Um sinal PWM pode ser visto como sinal PWM com média igual a zero ao qual foi adicionado uma tensão contínua (DC). Essa tensão contínua é igual ao valor médio do sinal PWM original, fato que já usamos para acionar nossos motores. A Figura 7.52 apresenta, de forma gráfica, este conceito.



Figura 7.52. Decomposição de um sinal PWM em uma componente DC e uma onda retangular de média zero.

A ideia é aproveitar o valor DC (valor médio) do PWM para ser a saída do DAC. Para tanto, vamos usar um filtro passa-baixa para filtrar (remover) a onda quadrada e deixar passar apenas o valor médio. Conseguimos essa remoção usando um filtro muito simples que é o filtro formado por um resistor ( $R$ ) e um capacitor ( $C$ ). A frequência de corte de tal filtro é dada por  $1/RC$  rad/s. Na Figura 7.53, sugerimos o filtro com  $R = 10\text{ k}\Omega$  e  $C = 100\text{ }\mu\text{F}$ , o que resulta na frequência de corte de  $1.000\text{ rad/s}$ . Para obtermos a frequência de corte em Hz, dividimos  $1.000$  por  $2\pi$ , e obtemos aproximadamente a  $160\text{ Hz}$  ( $159,16\text{ Hz}$ ).

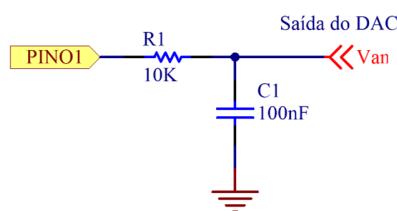


Figura 7.53. Sugestão de um filtro para remover a “onda retangular” de um PWM gerado na saída “pino1”. “Van” é a saída analógica do DAC.

Usualmente, a resposta em frequência de um filtro é expressa em dB. A equação abaixo indica como se calcula o ganho “ $g$ ”, em dB.

$$g_{dB} = 20 \times \log_{10}(g)$$

No texto acima, caracterizamos o filtro pela sua frequência de corte. Entretanto, é preciso deixar claro que, apesar do nome, frequência de corte indica apenas a frequência onde o ganho é igual a -3 dB, ou seja, ganho igual a  $0,707$  ( $1/\sqrt{2}$ ). O gráfico da Figura 7.54 apresenta a resposta em frequência para o filtro sugerido. Note que a escala em Hz (na horizontal) é logarítmica e que o ganho (escala vertical) está em dB. Para que a figura fique clara, o eixo horizontal vai de 10 Hz até 20 kHz. É fácil de ver que a frequência de corte é igual a 159 Hz. Vamos notar também que em 16 kHz o ganho é de -40 dB (ganho =  $1/100$ ).

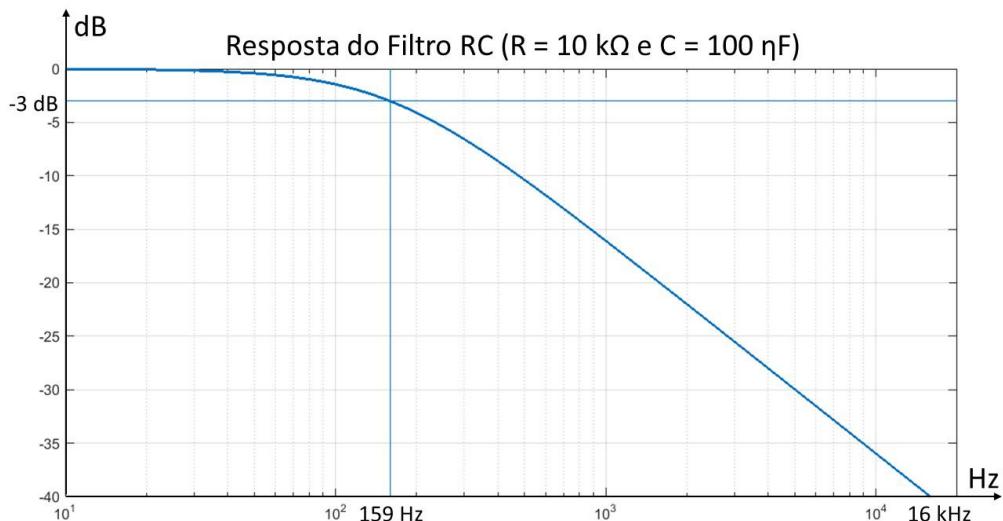


Figura 7.54. Resposta do filtro RC sugerido na Figura 7.53.

Precisamos especificar os parâmetros do PWM para gerar o DAC. É claro que o DAC vai trabalhar na faixa de 0 a 3,3 V. O primeiro ponto é que quanto mais alta for a frequência do PWM, maior será a atenuação da onda quadrada, por ação do filtro RC. A resolução do DAC é outro ponto importante, ou seja, quantos passos entre o valor mínimo e o máximo. É claro que o máximo é a alimentação de 3,3 V. Se pensarmos em 100, a resolução deste ADC será de 33 mV ( $3,3V / 100$ ). Parece pouco? Podemos escolher uma resolução melhor, com 1000 passos, cada um de 3,3 mV ( $3,3V / 1.000$ ).

Chegamos finalmente ao pedido. Escreva o código criar um conversor DAC usando o recurso de PWM de TB0.5 (P3.5). A frequência sugerida para o PWM deve ser de 20 kHz, o que garante uma boa filtragem e usar 1.000 passos (resolução) para ir de 0% até 100%. Será que com o MSP430 é possível gerar tal PWM? Verifique! Caso negativo, faça uma proposta. O *timer* B foi escolhido para aproveitar a dupla buferização. O controle para o

ensaio será feito com as chaves S1 (P2.1) e S2 (P1.1) e com os *leds* vermelho (P1.0) e verde (P4.7), segundo a tabela abaixo. Com o experimento funcionando e um voltímetro, levante a curva de saída do DAC.

*Tabela 7.19. Controle e sinalização para o ER 717*

Dispositivo	Ação
S1 (A → F)	+10% (33 mV)
S2 (A → F)	-10% (33 mV)
Verde	Inverte a cada acionamento de S1 ou S2
Vermelho	Aceso em 0% ou 100%

**Solução:** Vamos iniciar conferindo a possibilidade de criar o PWM sugerido. Foi proposto período de 20 kHz e 1.000 passos. Se queremos 1.000 passos dentro de uma janela de 20 kHz, a frequência do timer deve ser de, pelo menos, 20 MHz ( $1.000 \times 20$  kHz). Evidentemente não vamos conseguir, já que nosso relógio mais rápido é o SMCLK trabalha com, aproximadamente, 1 MHz (1.048.576 Hz). A melhor solução seria alterar o valor do SMCLK para 20 MHz, o que pode ser feito facilmente com a reprogramação da UCS, mas não é nosso objetivo neste capítulo.

Vamos então propor novos valores. A frequência do PWM em 10 kHz e 100 passos, de 0% até 100%. Passamos então a necessitar de um relógio de 1 MHz, que pode ser atendido pelo SMCLK. A atenuação do filtro em 10 kHz é de, aproximadamente, -22 dB (ganho = 1/12,6), de acordo com a Figura 7.54. É o que foi possível conseguir.

Vamos então colocar o contador TB0R para operar no Modo 1 (Up) e, por simplicidade, limitar a contagem em 100. Isso vai nos dar um PWM com frequência indicada pela conta abaixo. Fizemos a divisão por 101 e não por 100 porque o zero também é contado.

$$f_{PWM} = \frac{1.048.576}{101} = 10.382 \text{ Hz (aprox)}$$

Agora, fica simples o controle. Fazemos TB0CCR0 = 100 e em TB0CCR5 programamos o Ciclo de Carga com valores de 0% até 100%. A listagem do programa é apresentada logo a seguir.

#### Listagem solução do ER 7.17

```
// ER 7.17
// DAC construído com TB0.5 (P3.5)
// S1 aumenta em 10%,
```

```

// S2 diminui em 10%, Led verde aceso se PWM = 0%
// Led vermelho aceso se PWM = 0% ou 100%
// Led verde = inverte de estado a cada acionamento de S1 ou S2

#include <msp430.h>

#define TRUE 1
#define FALSE 0

#define ABT 1          //Constante representa Aberta
#define FEC 0          //Constante representa Fechada
#define DBC 1000       //Atraso para o debounce

#define T100us 100    //Período para 10 kHz
#define PASSO 1        //Passo de 1%

void tb0_config(void);      //Configuarar TB0.6
void config_pinos(void);   //Config Pinos
int check_s1(void);        //Checar chave S1
int check_s2(void);        //Checar chave S2
void debounce(void);        //Delay para debounce

int ps1=ABT,ps2=ABT;        //Estado anterior das chaves

int main(void)
{
    int pot=0;
    WDTCTL = WDTPW | WDTHOLD;    // stop watchdog timer
    config_pinos();             //Config pinos
    tb0_config();               //Config TB0
    while (TRUE) {
        if (check_s1()) {pot+=10; P4OUT ^= BIT7; }
        if (check_s2()) {pot-=10; P4OUT ^= BIT7; }
        if (pot < 0)    pot=0;        //Limite 0%
        if (pot > 100)  pot=100;     //Limite 100%
        TB0CCR5 = pot;            //Recalcular
        P1OUT &= ~BIT0;           //Apagar vermelho
        if (pot==100 || pot==0)   P1OUT |= BIT0; //Acender vermelho
    }
    return 0;
}

// Configurar TB0.6
void tb0_config(void) {
    TB0CTL = TBSSEL_2 | MC_1;      //SMCLK e MC=1
    TB0CCTL0= 0;
    TB0CCR0=T100us;              //Período de 1 ms (Modo 3)
    TB0CCTL5= CLLD_2 | OUTMOD_6;  //P3.5 = PWM para DAC
    TB0CCR5=0;                   //Iniciar com Ciclo de Carga 0%
    //Configurar pino P3.5 para PWM
}

```

```

P3DIR |= BIT5;          //TB0.5 = P3.5
P3SEL |= BIT5;          //Usar com PWM
}
int check_s1(void){...}    //Copiar do ER 7.8
int check_s2(void){...}    //Copiar do ER 7.8
void config_pinos(void){...} //Copiar do ER 7.8
void debounce(void){...}    //Copiar do ER 7.8

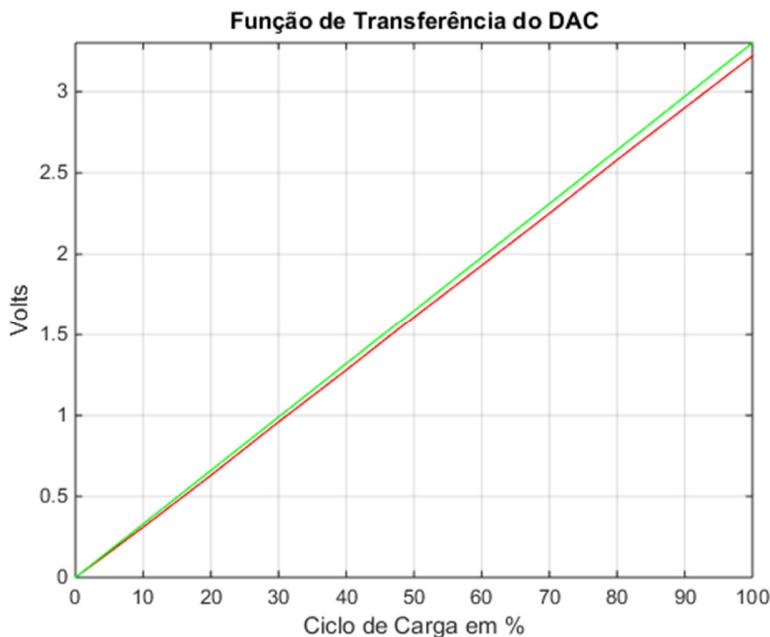
```

A seguir apresentamos a tabela com as medições feitas com o DAC proposto e sua função de transferência na Figura 7.55. É interessante observar como ele ficou preciso.

*Tabela 7.20. Medições feitas com o ADC proposto (erro = medido – ideal)*

Carga	Medido	Ideal	Erro
0%	0,00	0,00	0,00
10%	0,31	0,33	-0,02
20%	0,63	0,66	-0,03
30%	0,96	0,99	-0,03
40%	1,28	1,32	-0,04
50%	1,61	1,65	-0,04

Carga	Medido	Ideal	Erro
60%	1,93	1,98	-0,05
70%	2,25	2,31	-0,06
80%	2,58	2,64	-0,06
90%	2,90	2,97	-0,07
100%	3,22	3,30	-0,08



*Figura 7.55. Função de transferência do DAC proposto, segundo a Tabela 7.20. A linha verde corresponde à curva ideal.*

**ER 7.18.** Teste o DAC do exercício anterior num ambiente para gerar senoides de diferentes frequências, na faixa de 10 Hz até 200 Hz, e controladas por S1 e S2.

*Tabela 7.20. Controle e sinalização para o ER 7.18*

Dispositivo	Ação
S1 (A → F)	+10 Hz
S2 (A → F)	-10 Hz
Verde	Inverte a cada acionamento de S1 ou S2
Vermelho	Acesso nos valores extremos de frequência (10 Hz e 200 Hz)

**Solução:** Na solução do exercício anterior construímos um conversor DAC com 100 passos de Ciclo de Carga. O 0% corresponde a 0,0 V e o 100% a 3,3 V. Vamos agora gerar uma função seno usando esse recurso. Na biblioteca “Math” do CCS existe a função `sin(ang)` que calcula o seno, sendo que o parâmetro `ang` deve estar em radianos. Vamos usar essa função para criar um vetor com todos os valores que precisamos da função seno, ao longo do período ( $2\pi$ ).

Para facilitar os diversos ensaios, definimos no programa as três constantes abaixo. Elas permitem especificar a senoide com `SENO_PT` pontos por período ( $2\pi$ ), excursionando entre o mínimo `SENO_VMIN` e o máximo `SENO_VMAX`. Veja que na listagem apresentada está especificado uma senoide com 100 pontos e excursionando em 0,5 V e 3,3 V.

- `SENO_PT` → quantidade de pontos ao longo de um período da função
- `SENO_VMAX` → valor máximo da função em Volts, não ficar acima de 3,3V
- `SENO_VMIN` → valor mínimo da função em Volts, não ficar abaixo de 0,0 V

Os parâmetros “Amplitude” e “Deslocamento<sub>Vertical</sub>” da função seno são facilmente calculados com as duas equações abaixo.

$$\text{Amplitude} = \frac{100 \times (\text{SENO\_VMAX} - \text{SENO\_VMIN})}{2 \times 3,3}$$

$$\text{Deslocamento}_{\text{Vertical}} = \frac{100 \times \text{SENO\_VMIN}}{3,3}$$

Para calcular todos os pontos da função seno, vamos varrer a faixa  $[0, 2\pi]$  usando a equação abaixo. Na listagem, este trecho está marcado em cinza.

$$\text{seno}[i] = \text{Amplitude} \times \sin(i * \text{passo}) + \text{Amplitude} + \text{Deslocamento}_{\text{Vertical}}$$

Deixando a trigonometria de lado, vejamos a algumas considerações do programa. Geramos um vetor com os valores do seno na faixa  $[0, 2\pi]$ , de acordo com a resolução pedida. Precisamos agora atualizar o DAC na taxa ditada pela frequência do seno. Simplificar a explicação, vamos considerar o caso de resolução de 100 pontos (`SENO_PT`) e frequência de 100 Hz ( $T = 10$  ms). Isto significa que temos de atualizar o DAC numa taxa de  $10\text{ms}/100$ , ou seja, na frequência de 10 kHz. Para 200 Hz, a frequência de atualização é de 20 kHz.

Vamos usar o TA2.0 para interromper o programa e atualizar o DAC. Para tanto, lançamos mão da solução adotada no ER 7.4 e apresentada na página 465 do manual “MSP 430 – User Guide”, (item 17.2.3.3 *Use of Continuous Mode*). De acordo com essa solução, a cada interrupção de TA2CCR0, somamos o valor do passo (`ta2_passo`), marcando o instante da próxima interrupção. A equação abaixo mostra como calcular esse valor, sendo `freq`, a frequência da função seno.

$$\text{ta2\_passo} = \frac{\text{SMCLK} = 1.048.576}{\text{freq} \times \text{SENO\_PT}}$$

O ensaio do programa solução mostrou um limite na frequência do sinal gerado. Com 100 pontos de resolução não se consegue gerar seno acima de 170 Hz. A explicação é simples. Nessas condições, o passo de TA2CCR0 é de 61 contagens de SMCLK, o que corresponde, aproximadamente, a um intervalo de 58  $\mu\text{s}$ . Isto significa que um intervalo de 58  $\mu\text{s}$  é muito pequeno para o MSP atender a interrupção para atualizar o DAC. Em outras palavras, é preciso de mais de 58  $\mu\text{s}$ .

Temos duas saídas. Uma é diminuir a resolução (`SENO_PT`) da senoide e a outra é aumentar o relógio (MCLK = 1.048.576 Hz) da CPU. Quando estudarmos DMA, veremos que existe uma terceira solução muito mais elegante.

As duas figuras abaixo mostram ensaios em 100 Hz e em 170 Hz com 100 pontos de resolução. Na Figura 7.56 é possível de ver que o seno gerado ficou um pouco abaixo do limite que era de [0,5 V até 2,5 V]. À medida que se abaixa a frequência do seno, aproxima-se dos valores especificados.

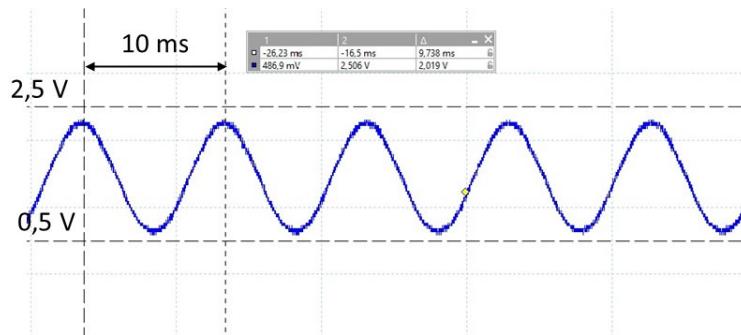


Figura 7.56. Resultado na geração da função seno de 100 Hz com 100 pontos de resolução.

Já na Figura 7.57 é possível de ver que há problemas em 170 Hz. O tempo gasto pela interrupção é maior que a taxa de atualização exigida. Note os platôs horizontais que indicam que o DAC ficou bastante tempo sem ser atualizado.

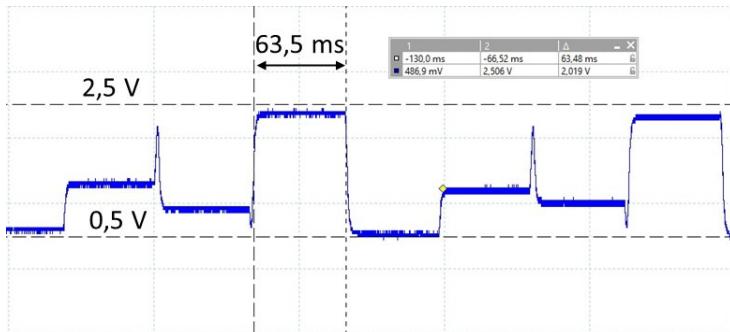


Figura 7.57. Problemas na geração da função seno de 170 Hz com 100 pontos de resolução.

#### Listagem solução do ER 7.18

```
// ER 7.18
// Gerar senoides com diversas frequências
// DAC construído com TB0.5 (P3.5)
// S1 soma FREQ_PASSO na frequência da senoide
// S2 subtrai FREQ_PASSO da frequência da senoide
// Led vermelho aceso se frequência = FREQ_MIN ou FREQ_MAX
// Led verde = inverte de estado a cada acionamento de S1 ou S2

#include <msp430.h>
#include <math.h>    //para a função sin()

#define TRUE 1
```

```

#define FALSE 0

#define ABT 1      //Constante representa Aberta
#define FEC 0      //Constante representa Fechada
#define DBC 1000   //Atraso para o debounce

#define SMCLK     1048576L
#define T100us    100 //Período para 10 kHz
#define SENO_PT    100 //Qtd de pontos para função seno
#define SENO_VMAX  2.5 //Tensão máximo para o seno
#define SENO_VMIN  0.5 //Tensão mínima para o seno
#define PI         3.14159 //Sem comentários
#define FREQ_MIN   10 //Freq mínima aceita
#define FREQ_MAX   200 //Freq máxima aceita
#define FREQ_PASSO 10 //Passa para excursionar freq

void seno_calc(void);           //Criar vetor seno
void ta2_config(void);          //Configurar TA2.0
void tb0_config(void);          //Configuarar TB0.6
void config_pinos(void);        //Config Pinos
int check_s1(void);            //Checar chave S1
int check_s2(void);            //Checar chave S2
void debounce(void);            //Delay para debounce

int ps1=ABT,ps2=ABT;           //Estado anterior das chaves
volatile char seno[SENO_PT];   //Vetor com a função seno
volatile unsigned int ix;       //Indexar vetor seno
volatile unsigned int ta2_passo; //Passo para TA2CCR0

int main(void){
    int freq=FREQ_MIN;
    WDTCTL = WDTPW | WDTHOLD;    // stop watchdog timer
    seno_calc();                //Construir vetor seno
    config_pinos();              //Config pinos
    tb0_config();                //Config TB0
    ta2_config();                //Config TA2
    ta2_passo=SMCLK/freq;        //Passo inicial para TA2CCR0
    __enable_interrupt();
    while (TRUE){
        if (check_s1()) {freq+=FREQ_PASSO; P4OUT ^= BIT7; }
        if (check_s2()) {freq-=FREQ_PASSO; P4OUT ^= BIT7; }
        if (freq < FREQ_MIN)   freq=FREQ_MIN;           //Limite Inferior
        if (freq > FREQ_MAX)   freq=FREQ_MAX;           //Limite Superior
        ta2_passo=SMCLK/(SENO_PT*(long)freq);           //Recalcular passo
TA2CCR0
        P1OUT &= ~BIT0;                                //Apagar vermelho
        if (freq==FREQ_MIN || freq==FREQ_MAX)
            P1OUT |= BIT0; //Acender vermelho
    }
    return 0;
}

```

```

}

// Construir vetor seno com 100 pontos
void seno_calc(void){
    unsigned int i;
    float ang, passo, aux;
    volatile unsigned int amp, bias;
    amp=round(100*((SENO_VMAX-SENO_VMIN)/3.3));
    amp = amp/2;                                //Amplitude
    bias=round(100*(SENO_VMIN/3.3));           //Desloc vertical
    passo=(2*PI)/SENO_PT;                      //Resolução
    ang=0;
    for (i=0; i<SENO_PT; i++){
        aux=amp*sin(ang)+amp+bias;
        seno[i]=round(aux);
        ang+=passo;
    }
}

// Interrupção TA2.0
// #pragma vector = TIMER2_A0_VECTOR
#pragma vector = 44
__interrupt void ta2_ccr0(void){
    TB0CCR5=seno[ix++];           //Programar DAC
    if (ix==SENO_PT) ix=0;        //Limite indexador
    TA2CCR0 += ta2_passo;         //Proxima interrupção
}

// Configurar TA2.0 para gerar freq
void ta2_config(void){
    TA2CTL = TASSEL_2 | MC_2;      //SMCLK e Modo contínuo
    TA2CCTL0 = CCIE;              //Interrupção
    TA2CCR0=0;
}
void tb0_config(void){...}     //Copiar do ER 7.17
int check_s1(void){...}        //Copiar do ER 7.8
int check_s2(void){...}        //Copiar do ER 7.8
void config_pinos(void){...}   //Copiar do ER 7.8
void debounce(void){...}       //Copiar do ER 7.8

```

## 7.6. Exercícios Propostos

Apresentamos a seguir uma lista com diversos exercícios para que o leitor pratique o que foi estudado. Sempre que possível, verifique sua solução no LaunchPad.

**EP 7.1.** Usando um dos relógios do MSP faça o led verde (P4.7) piscar na taxa de 10 Hz. A cada 10 “piscadas” do led verde, o led vermelho inverte de estado. Apresente duas soluções: uma usando *polling* e a outra usando interrupções.

**EP 7.2.** Escreva um programa que gere pelos pinos P8.1 e P8.2 os dois sinais indicados abaixo. Tente a solução usando apenas um *timer*. Em seu programa, procure por simplicidade no código.

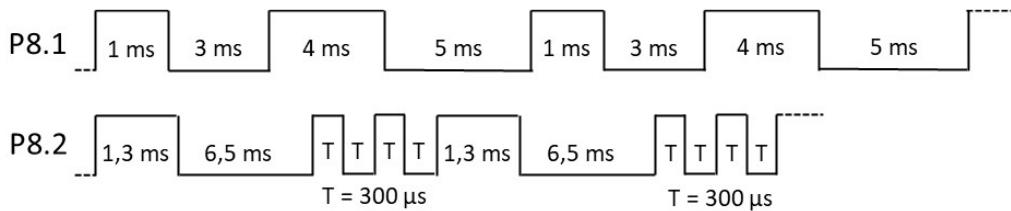


Figura 7.58. Diagrama de tempo dos sinais a serem gerados pelos pinos P8.1 e P8.2. O desenho não está em escala de tempo.

**EP 7.3.** Aproveitando um dos relógios apresentados no ER 7.3, adicione o recurso de alarme. Quando chegar o instante programado, os dois *leds* devem piscar de forma complementar, na frequência de 10 Hz. O acionamento de uma das chaves, S1 ou S2, desliga o alarme.

**EP 7.4.** Aproveitando o EP 7.3, prepare um relógio com 3 alarmes, com o seguinte comportamento.

- Alarme 1: pisca *led* verde em 10 Hz e para (**pára**) com o acionamento de S2;
- Alarme 2: pisca *led* vermelho em 10 Hz e para (**pára**) com o acionamento de S1;
- Alarme 3: pisca de forma complementar ambos os *leds* em 10 Hz e para (**pára**) com o acionamento de qualquer uma das chaves.

**Opinião:** Nunca deveríamos ter removido o acento diferencial entre o verbo **parar** e a preposição **para**.

**EP 7.5.** Vamos controlar o pisca-pisca dos *leds* vermelho (P1.0) e verde (P4.7) com as duas chaves S1 (P2.1) e S2 (P1.1) de acordo com a tabela abaixo.

Tabela 7.21. Padrão de pisca-pisca dos *leds* para o EP 7.5

S1 (P2.1)	S2 (P1.1)	Piscar Led vermelho	Piscar Led verde
Aberta	Aberta	3 Hz	7 Hz
Aberta	Fechada	3 Hz	10 Hz

Fechada	Aberta	10 Hz	7 Hz
Fechada	Fechada	10 Hz	10 Hz

**EP 7.7.** No ER 7.8 usamos o Ciclo de Carga de um PWM para controlar, de forma complementar, o brilho dos dois *leds*. O leitor deve ter tido a sensação de que a variação do brilho não era linear. O *led* parecia “chegar muito cedo” ao brilho máximo. Isso acontece porque nosso olho não apresenta uma resposta linear a variações de brilho, mas sim logarítmica. Uma forma de compensar esta resposta é fazer o brilho do *led* variar de forma exponencial. Assim, nosso olho terá a sensação de uma variação linear no brilho. Apresentamos na Figura 7.59 uma sugestão para se criar uma resposta exponencial. O eixo horizontal está graduado em 100 passos e o eixo vertical indica (em %) o Ciclo de Carga do PWM. Refaça o ER 7.8, mas agora garantindo a sensação de variação linear do brilho do *led*.

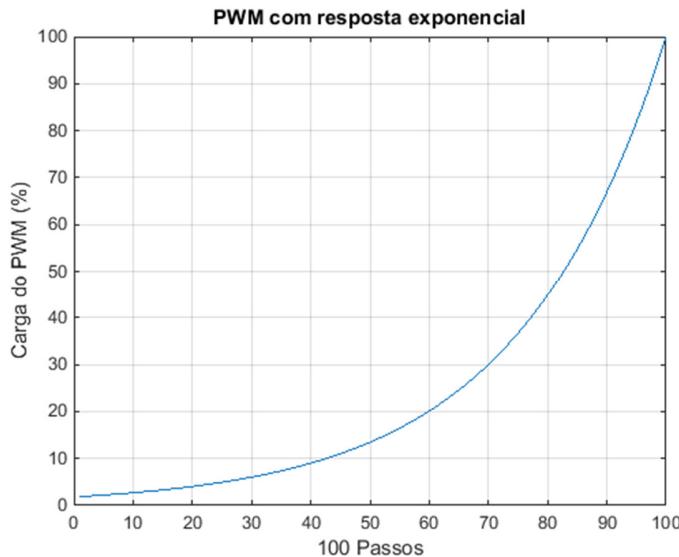


Figura 7.59. Sugestão para o controle exponencial do ciclo de carga de um PWM.

**EP 7.8.** Neste exercício vamos fazer o uso de um *buzzer* passivo (sem o oscilador interno) ou de um amplificador com alto-falante (vide Figura 7.60), conectado em P3.5 (TB0.5) para gerar tons em diversas frequências, controlados pelas chaves S1 e S2 e usando os *leds* para auxiliar na sinalização da faixa de frequência. O programa parte gerando tom em 0 Hz. De acordo com o indicado na tabela, o limite deste exercício é de 7 kHz. O leitor pode estender este limite. Entretanto, é provável que o transdutor já não apresente bom desempenho.

- Cada acionamento de S1 (P2.1) → +200 Hz
- Cada acionamento de S2 (P1.1) → -200 Hz

Tabela 7.22. Padrão de pisca-pisca dos leds para o EP 7.5

Led Vermelho (P1.0)	Led Verde (P4.7)	Frequência Do tom (Hz)
Apagado	Apagado	$0 \leq \text{tom} < 1\text{kHz}$
Apagado	Aceso	$1\text{kHz} \leq \text{tom} < 2\text{kHz}$
Aceso	Apagado	$2\text{kHz} \leq \text{tom} < 3\text{kHz}$
Aceso	Aceso	$3\text{kHz} \leq \text{tom} < 4\text{kHz}$
Apagado	Piscando (10Hz)	$4\text{kHz} \leq \text{tom} < 5\text{kHz}$
Piscando (10Hz)	Apagado	$5\text{kHz} \leq \text{tom} < 6\text{kHz}$
Piscando (10Hz)	Piscando (10Hz)	$6\text{kHz} \leq \text{tom} < 7\text{kHz}$

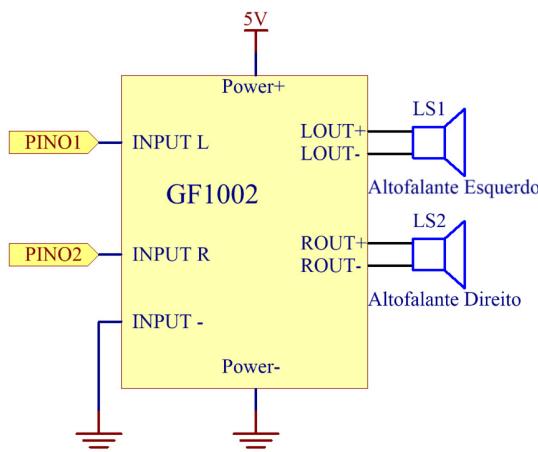


Figura 7.60. Sugestão para conexão do amplificador GF1002 ao MSP. O leitor pode usar apenas um dos canais (esquerdo ou direito).

**EP 7.9.** Vamos repetir o EP 7.8, mas agora a variação de frequência será automática. Deverá ser varrida a faixa ascendente de 0 Hz até 10 kHz e em seguida a faixa descendente de 10 kHz até 0 Hz. Depois, tudo se repete. Em seu programa, o leitor deverá especificar com uma constante, o tempo para o programa completar um ciclo (0 Hz → 10 kHz → 0 Hz). Faça vários ensaios alterando esta constante. Fica a critério do leitor decidir qual o tamanho dos passos.

```
#define TCICLO 10 //Realiza o ciclo em 10 segundos
```

Os *leds* deverão fazer, de forma autônoma, a sinalização indicada na tabela a seguir.

*Tabela 7.23. Sinalização dos leds para o EP 7.9*

Led Vermelho (P1.0)	Led Verde (P4.7)	Frequência Do tom (Hz)
Apagado	Apagado	$0 \leq \text{tom} < 2,5 \text{ kHz}$
Apagado	Aceso	$2,5 \text{ kHz} \leq \text{tom} < 5,0 \text{ kHz}$
Aceso	Apagado	$5,0 \text{ kHz} \leq \text{tom} < 7,5 \text{ kHz}$
Aceso	Aceso	$7,5 \text{ kHz} \leq \text{tom} \leq 10 \text{ kHz}$

**EP 7.10.** Este exercício é para os leitores que entendem um pouco de música ou que têm disposição. Aproveitando a estrutura do ER 7.10, programe outras músicas. Sugestões:

- Tema do Jornada nas Estrelas (melhor que Star Wars!);
- Trecho mais conhecido do Tico-Tico no Fubá;
- Refrão do Mamãe Eu Quero e
- Trecho da Melodia do “Caneta Azul” (só para os bravos, fortes e filhos do norte).

**EP 7.11.** Agora sim, este é só para os entendidos em música. Com o amplificador de dois canais, sugerido na Figura 7.60, podemos gerar acordes com duas notas diferentes. Fica a ideia para o leitor que estiver disposto a codificar de forma um pouco mais rica algum trecho de música.

**EP 7.12.** Vamos trabalhar com o servomotor apresentado no ER 7.11. O problema pede para controlar o servo de modo que ele execute, repetidamente, um ciclo de 360 graus num determinado período. Por ciclo de 360 graus se entende a excursão de  $-90 \rightarrow +90 \rightarrow -90$  graus. O período da excursão será controlado pelas chaves S1 (P2.1) e S2 (P1.1), de acordo com a tabela abaixo. Note que não há limite inferior. O valor inicial do período é de 5 segundos. Tente refazer este exercício testando o funcionamento do servo com períodos menores que 1 segundo.

*Tabela 7.24. Controle do período feito pelas chaves*

Dispositivo	Ação
S1 (A → F)	+1 seg
S2 (A → F)	-1 seg

**EP 7.13.** À solução do problema anterior, acrescente uma sinalização com os leds. O brilho do vermelho (P1.0) vai de 0% a 100% quando o servo excursiona na faixa de  $0 \rightarrow -$

90 graus. Já o brilho do led verde (P4.7) varia de 0% a 100% quando o servo excursiona na faixa de 0 → 90 graus.

**EP 7.14.** Usando o que foi estudado no ER 7.12, acione um motor DC de forma que sua velocidade varie do estado parado (PWM = 0%) para o de rotação máxima (PWM = 100%) e volte ao estado parado (PWM = 0%), continuamente, em um determinado período. O período dessa excursão será controlado pelas chaves S1 (P2.1) e S2 (P1.1), de acordo com a Tabela 7.24 (exercício anterior). Note que não há limite inferior. O valor inicial do período é de 5 segundos. Tente refazer este exercício testando o funcionamento do motor com períodos menores que 1 segundo.

**EP 7.15.** Com uso de uma Ponte H, podemos controlar a velocidade e direção de um motor DC. Repita o EP 7.14, mas agora fazendo a rotação do motor excursionar desde o valor máximo no sentido direto até o valor máximo no sentido reverso e voltando ao máximo no sentido direto. Veja o tópico M.2.1 do Apêndice M e o ER M.2.

**EP 7.16.** Aqui é pedido para construir um jogo para ver qual chave é acionada primeiro e em quanto tempo ela antecedeu a outra. O jogo começa com os dois *leds* acesos. Depois de um tempo de aproximado de 5 segundos, os *leds* apagam, indicando o momento de acionar as chaves. Se a chave S1 for acionada primeiro, *led* vermelho (P1.0) acende, do contrário o *led* verde (P4.7) acende. O tempo em que uma chave antecedeu a outra deve ser indicado na variável `float tempo` (em microssegundos). O jogo só recomeça com as duas chaves liberadas. Para gerar um atraso de 5 segundos, use a função `_delay_cycles(5000000)`.

*Tabela 7.25. Conexão das chaves com os timers*

Chave	Pino	Timer
S1	P2.1	TA1.2
S2	P1.1	TA0.0

**EP 7.17.** Aqui é pedido para construir um jogo onde o jogador deve acionar e liberar a chave S1 cinco vezes, sendo que os períodos com a chave fechada e aberta devem ser o mais próximos possível. Ao final dos cinco acionamentos, o programa deve apresentar nas variáveis `maior` e `menor`, o maior e menor período, em contagens do SMCLK. Na variável `dif` deve apresentar a diferença entre esses dois valores. Ganha quem conseguir a menor diferença. Cuidado com os rebotes.

**EP 7.18.** Com os mesmos recursos do ER 7.17, use o DAC proposto para gerar um sinal dente de serra com frequência variável.

**EP 7.19.** Repita o exercício anterior, mas agora gerando um sinal triangular.

**EP 7.20.** No ER 7.16 criamos uma forma de se medir a velocidade de rotação do motor usando um sensor infravermelho. Esta forma de medição é muito imprecisa e sujeita a erros. É comum surgir pulsos espúrios que levam a erros na medida de velocidade. Precisamos de uma forma de se eliminar esses pulsos equivocados. Uma ideia é criar uma função juiz que recebe o vetor com as diferenças e elimina os valores discrepantes, por exemplo, com variação acima de 20%. Refaça o ER 7.16, mas agora adquirindo um vetor com 32 diferenças e depois usando uma função juiz para eliminar os valores discrepantes. A estimativa da velocidade será feita somente com os valores considerados coerentes. Será que sob essas novas condições a medição de velocidade ficou mais estável? É recomendado dar uma olhada no Apêndice M, ER M.01. Veja que lá se caracterizou um perfil para os pulsos válidos.

**EP 7.21.** Tendo uma forma um pouco mais confiável de se medir a rotação de um motor, podemos propor um controle de velocidade em malha fechada. Até agora, o controle de velocidade que fizemos variando o Ciclo de Carga do PWM só funciona com o motor girando sem carga. Se o motor tiver de acionar algo, por exemplo, as rodas de um pequeno carro, essa velocidade irá cair. Vamos então propor uma melhoria. Faremos um controle em malha fechada, como mostrado na Figura 7.61. O programa vai aumentar ou diminuir a carga do PWM de forma a manter a velocidade especificada pelo usuário.

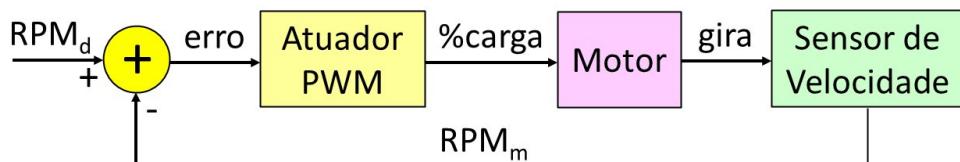


Figura 7.61. Sugestão de um esquema em malha fechada para controlar a velocidade de rotação do motor.  $RPM_d$  indica o valor desejado e  $RPM_m$  indica o que foi medido.

Na figura acima, o valor especificado (desejado) pelo usuário foi rotulado por  $RPM_d$ . O valor medido é indicado por  $RPM_m$ . Temos então o sinal de erro, dado pela diferença entre os dois valores:  $erro = RPM_d - RPM_m$ . O Atuador PWM vai variar a carga do PWM em função desse erro. Se o erro aumentar, será preciso aumentar o Ciclo de Carga do PWM. A sugestão é partir com um Ciclo de Carga calculado para o motor sem carga, que passamos a chamar de  $CC_b$  (básico) e, a partir daí, aumentarmos ou diminuirmos o valor do Ciclo de Carga ( $CC$ ) em função da variação da rotação do motor. O nome dado a este processo é o de Controlador Proporcional, já que ele atua em função da proporcionalidade do erro. A expressão abaixo expressa o conceito de forma clara.

$$CC = CC_b + K_p \times erro$$

Existem formas de se calcular  $K_p$ , porém fogem ao objetivo deste livro. Assim, o leitor deverá, de forma empírica, tentar descobrir o valor ideal da Constante de Proporcionalidade ( $K_p$ ). Faça alguns ensaios imaginários para ter uma primeira estimativa de  $K_p$ . Valores grandes de  $K_p$  podem levar o sistema a oscilar. Outro ponto importante, que será deixado para o leitor decidir, é a periodicidade da atuação. Deve-se a atuar a cada medida, a cada 10 medidas, a cada segundo ou a cada minuto?

### EP 7.22. CUIDADO NESTE EXERCÍCIO HÁ RISCO DE CHOQUE ELÉTRICO.

Se o leitor não tiver experiência em trabalhar com a rede elétrica, peça ajuda. Uma forma de se controlar a potência entregue a dispositivos ligados à rede elétrica tais como lâmpadas incandescentes, aquecedores elétricos ou motores CA, é com o emprego de tiristores. Neste exercício será empregado um tipo de tiristor denominado TRIAC. A figura a seguir apresenta uma sugestão para controle da potência entregue à carga RL com o emprego de um TRIAC (Q1 - TIC246). Como o TRIAC vai operar diretamente com a rede elétrica, é bom prever uma proteção para o resto do circuito, que no caso foi feita com o foto-TRIAC (Q2 - MOC3020), que oferece isolação óptica.

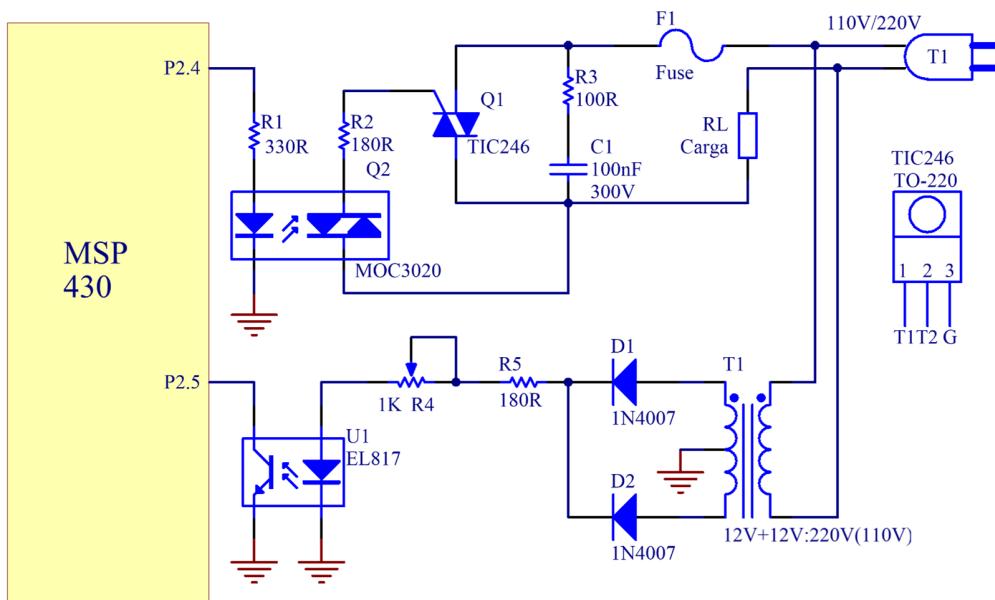
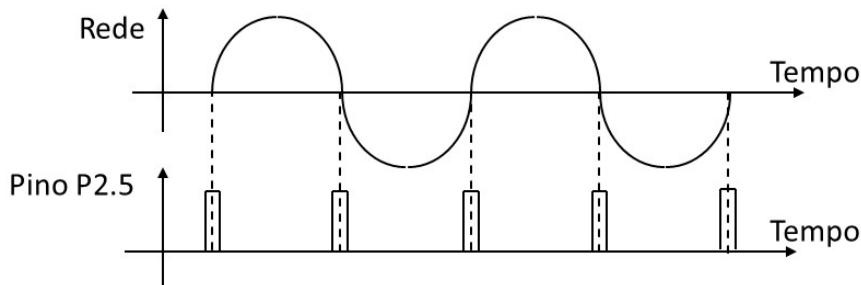


Figura 7.62. Sugestão de um circuito para o controle da potência entregue à carga RL.

Sempre que o pino P2.4 for colocado em nível alto, o foto-TRIAC Q2 conduz, o que leva à condição de disparo para o terminal de “Gate” do TRIAC Q1, que passa a conduzir. Se o TRIAC conduz, a energia da rede chega até a carga RL. Para eliminar um pouco os ruídos gerados pelo chaveamento do TRIAC, usa-se o filtro formado por C1 e R3. É importante que C1 seja um capacitor cerâmico capaz de suportar mais de 300 V.

Para que se tenha noção dos instantes em que se deve disparar o TRIAC, é necessária a sincronização com a rede elétrica. Uma maneira simples de se conseguir tal sincronização é através de um circuito que detecte as passagens pelo zero, ou seja, um circuito que avise quando a tensão alternada da rede estiver passando pelo valor zero. Com essa finalidade vamos usar o pino P2.5, que vai ativar uma interrupção. Este pino deve ser configurado como entrada com o *pullup*.

Na figura acima, o transformador T1 com tomada central (*center tap*) e os diodos D1 e D2 fazem um retificador de onda completa. Enquanto houver tensão, o acoplador óptico U1 conduz e, com isso, tem-se um nível baixo no pino P2.5. Mas, quando a rede estiver passando pelo zero, não haverá tensão para acender o led do opto-acoplador e, com isso, o foto-transistor vai para o corte. Nesse caso, graças ao *pullup* interno, o pino P2.5 irá para nível alto. O resultado serão pulsos a cada passagem pelo zero, como ilustrado na Figura 7.63. O potenciômetro R4 permite um pequeno controle da largura desses pulsos.



*Figura 7.63. Detecção do zero da rede elétrica.*

O controle de potência é feito a cada semiciclo da rede elétrica, como mostrado na Figura 7.64. O instante de disparo do TRIAC, comandado através do pino P2.4, deve tomar como referência a passagem pelo zero (pino P2.5). Se o disparo for logo após a passagem pelo zero, como é o caso do pulso P1, a carga é alimentada durante quase que todo o semiciclo. Um disparo um pouco mais tarde, como acontece com P2, entrega um pouco menos de potência à carga. Por outro lado, se o disparo for muito tarde, como é o caso de P4, quase nenhuma potência chega até a carga. Desta forma, através do instante de disparo do TRIAC (pino P2.4), controla-se a potência que é entregue à carga. Por exemplo, o pulso P3 (atraso em relação ao zero igual a tp3) acontece aproximadamente na metade do semi-ciclo, com isso a carga recebe apenas 50% da potência total.

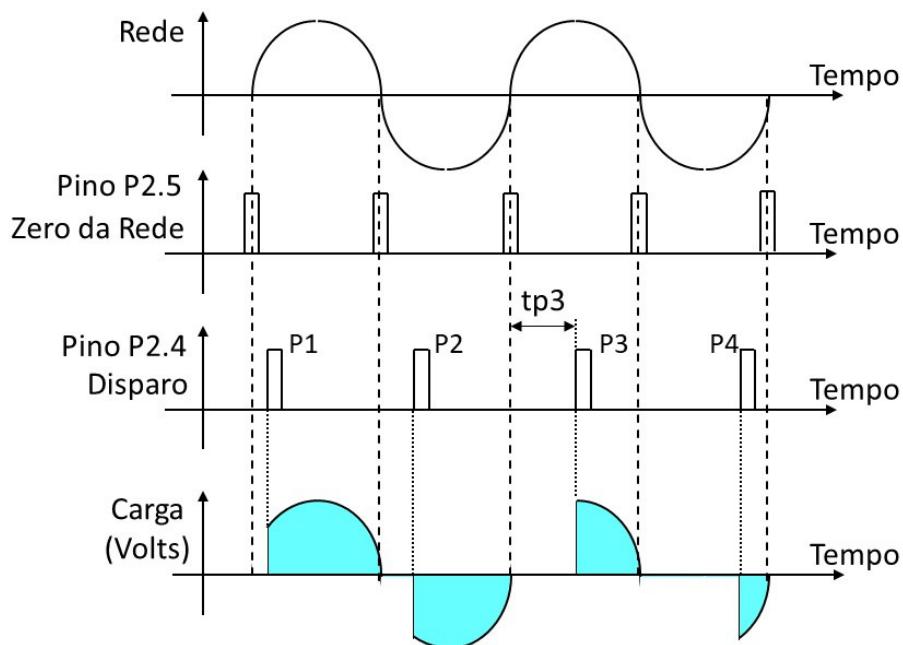


Figura 7.64. Quatro situações de disparo do TRIAC e a tensão entregue à carga.

Finalmente, chegamos ao pedido. Escreva uma rotina que inicie com a carga recebendo potência zero. Controlado pelo acionamento das chaves S1 (P2.1) e S2 (P1.1), a potência será aumentada ou diminuída, dentro de uma faixa de 0 a 100, como indicado na Tabela 7.26.

Tabela 7.26. Controle de potência e sinalização para o EP 7.22

Dispositivo	Ação
S1 (A → F)	+10% na carga
S1 (A → F)	-10% na carga
Led Vermelho	Aceso se carga igual a 100%
Led Verde	Aceso se carga igual a 0%

O semicírculo da rede tem a duração de 8,33 ms. Para simplificar, divida o semicírculo em 100 passos, ou seja, cada passo tem a duração de 83,3  $\mu$ s. Lembre-se que:

- Potência = 0%  $\rightarrow$  sem pulso de disparo;
- Potência = 25%  $\rightarrow$  pulso de disparo em 6,25 ms;
- Potência = 50%  $\rightarrow$  pulso de disparo em 4,17 ms;
- Potência = 75%  $\rightarrow$  pulso de disparo em 2,083 ms e
- Potência = 100%  $\rightarrow$  pulso de disparo logo após a passagem pelo zero

Observação: o pulso de disparo gerado pelo pino P2.4 não pode ser muito estreito; um valor entre 20 a 50  $\mu\text{s}$  já deve ser suficiente.

**EP 7.23.** No exercício anterior dividiu-se o semiperíodo em 100 intervalos de 83,3  $\mu\text{s}$ . Entretanto, cada um desses intervalos não corresponde a 1/100 da potência total. Refaça o exercício anterior recalculando os instantes de disparo de tal forma que se trabalhe realmente com incrementos de 1/100 da potência plena.

Sugestão: Para uma carga  $RL$  alimentada pela rede com frequência  $f$  e amplitude  $A$  e intervalo de condução de 0 a  $\varphi$ , a potência na carga é dada por:

$$P = \frac{A^2}{RL} \int_0^\varphi \sin^2 2\pi f t \cdot dt$$

**EP 7.24. CUIDADO NESTE EXERCÍCIO HÁ RISCO DE CHOQUE ELÉTRICO.**

Se o leitor não tiver experiência em trabalhar com a rede elétrica, peça ajuda. Hoje em dia temos a disponibilidade dos Relés de Estado Sólido (SSR – *Solid State Relay*) para acionar cargas AC. O funcionamento é análogo ao de um relé. Ao se acionar uma determinada entrada, o relé entra em condução. O relé corta quando se remove o acionamento desta entrada. Na maioria dos casos de emprego de tais SSR, não se pretende recortar o semiciclo. O controle é mais grosseiro, e não se faz a sincronização com a rede elétrica.

Este exercício pede o acionamento de uma carga AC com um SSR. Use o controle e sinalização especificados na Tabela 7.26. As duas figuras abaixo apresentam sugestões. A Figura 7.65 indica o circuito para SSR G3mb que é uma pequena placa que facilita o uso deste relé. Note que o acionamento acontece colocando o pino P2.4 em nível baixo. A Figura 7.66 sugere o circuito para o uso do SSR fabricado pela Fotek. Neste caso, não existe uma placa disponível, por isso, usa-se o próprio dispositivo. Os relés das Fotek são interessantes por alcançarem grande capacidade de corrente (10A, 20A, 50A, etc)

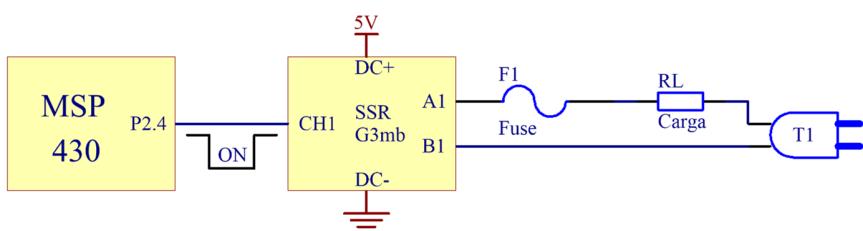


Figura 7.65. Exemplo de uso do SSR G3mb.

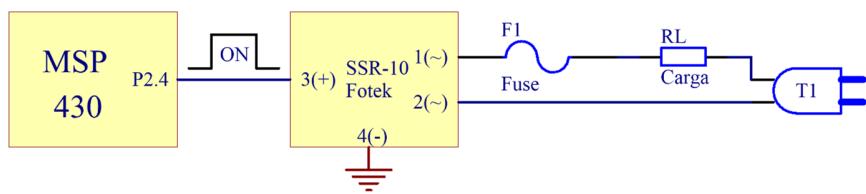


Figura 7.66. Exemplo de uso do SSR da Fotek.

Observação: Na solução deste exercício o leitor vai precisar escolher o período de seu PWM. Note que não há sincronização com a rede elétrica. Assim, podemos sugerir como período para o PWM o intervalo de tempo correspondente a 100 semiciclos (fica fácil construir 100 passos). Note que, como não há sincronização com a rede, há a incerteza de +/- 1 semiciclo em cada disparo.

*Gabarito para a configuração dos registradores das instâncias do Timer A*

<b>TAxCTL</b>	15	14	13	12	11	10	9	8
	-							TASSEL
	7	6	5	4	3	2	1	0
	ID		MC		-	TACLR	TAIE	TAIFG

<b>TAxCCTLn</b>	15	14	13	12	11	10	9	8
	CM		CCIS		SCS	SCCI	-	CAP
	7	6	5	4	3	2	1	0
	OUTMOD		CCIE		CCI	OUT	COV	CCIFG

<b>TAxEX0</b>	15 ... 3			2	1	0		
	-			TAIDEX				

<b>TAxR</b>	15 ... 0							

<b>TAxCCRn</b>	15 ... 0							

<b>TAxIV</b>	15 ... 0							

Gabarito para a configuração dos registradores das instâncias do Timer B

<b>TB0CTL</b>	15	14	13	12	11	10	9	8
	-	TBCLGRP0	CNTL		-	TBSSEL		
	7	6	5	4	3	2	1	0
	ID		MC		-	TBCLR	TBIE	TBIFG

<b>TB0CCTLn</b>	15	14	13	12	11	10	9	8
	CM		CCIS		SCS	CLLD		CAP
	7	6	5	4	3	2	1	0
	OUTMOD		CCIE	CCI	OUT	COV	CCIFG	

<b>TB0EX0</b>	15 ... 3	2	1	0
	-		TBIDEX	

<b>TB0R</b>	15 ... 0

<b>TB0CCRn</b>	15 ... 0

<b>TB0IV</b>	15 ... 0

