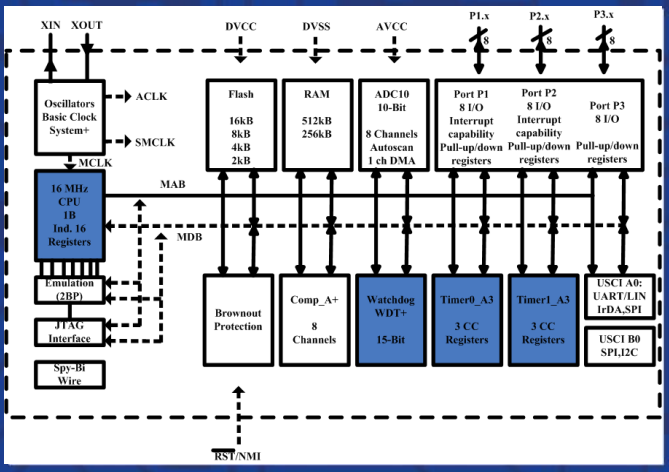
FRANCISCO EVERTON UCHÔA REIS

O MICROCONTROLADOR MSP430G2553

RECURSOS ESSENCIAIS: INTERRUPÇÕES, MODOS DE BAIXO CONSUMO E TEMPORIZADORES



MÓDULO II

edufpi

O Microcontrolador MSP430G2553

FRANCISCO EVERTON UCHÔA REIS

O Microcontrolador MSP430G2553

Recursos essenciais: Interrupções, modos de baixo consumo e temporizadores

MÓDULO II

edufpi

Projeto Gráfico. Capa. Diagramação.
Josafá Araújo Procópio

Revisão
Maria do Socorro Baptista Barbosa

FICHA CATALOGRÁFICA
Universidade Federal do Piauí
Biblioteca Comunitária Jornalista Carlos Castello Branco
Divisão de Representação da Informação

R375m Reis, Francisco Everton Uchôa.
O microcontrolador MSP430G2553: recursos essenciais: interrupções, modos de baixo consumo e temporizadores: módulo II / Francisco Everton Uchôa Reis.
– Teresina : EDUFPI, 2026.
64

ISBN 978-65-5904-504-4

1. Interrupções. 2. Microcontrolador. 3. Temporizadores. I. Título.

CDD 004.16

Bibliotecária: Francisca das Chagas Dias Leite – CRB3/1004

Editora da Universidade Federal do Piauí - EDUFPI
Campus Universitário Ministro Petrônio Portella
CEP: 64049-550 - Bairro Ininga - Teresina - PI - Brasil

CAPÍTULO 1 – INTERRUPÇÕES E MODOS DE BAIXO CONSUMO DE ENERGIA

1.1 Interrupções

A eficiência energética e respostas rápidas são características essenciais para microcontroladores e, para obter tais características, são utilizados conceitos como interrupções e modos de baixo consumo. O MSP430G2553 pode ser colocado em um modo de baixo consumo até o aparecimento de um evento. Diante da ocorrência do evento, o microcontrolador faz uso da interrupção para despertar do modo de baixo consumo e responder ao evento crítico, podendo ser a interrupção do tipo mascarável ou não mascarável. Esse microcontrolador usa interrupções vetorizadas, ou seja, cada interrupção possui seu próprio vetor, que está armazenado em um endereço predefinido em uma tabela de vetores.

A interrupção é um mecanismo que indica à CPU que um evento requer uma resposta imediata. Quando a interrupção é identificada, é feito o desvio da execução principal do código para executar uma sub-rotina chamada RTI (Rotina de Tratamento de Interrupção) daquela

fonte de interrupção solicitada. Após o término da execução, a CPU retoma a rotina normal das instruções do ponto anterior à interrupção.

1.1.1 Processo de interrupção

O processo de interrupção inicia-se quando um *bit* de *flag* de interrupção é setado e quando a condição para interrupção ocorre. A solicitação de uma interrupção é passada para a CPU se o bit GIE estiver definido e se o bit de habilitação individual daquela fonte de interrupção estiver habilitado. O atendimento dessa solicitação gera uma sequência de etapas que levam seis ciclos de *clock* antes que a RTI comece. Essas etapas são executadas na seguinte ordem:

1. Se a CPU estava ativa quando a interrupção foi solicitada, a instrução em execução no momento é completada;
2. O contador de programa (PC), que aponta para a próxima instrução é colocado na pilha, a fim de retornar ao ponto em que estava no programa;
3. O conteúdo do registrador SR é salvo na pilha;
4. A interrupção com mais alta prioridade é selecionada se múltiplas interrupções estiverem aguardando a execução;

5. A *flag* de interrupção é apagada automaticamente para vetores que possuem uma única fonte. Para interrupções com múltiplas fontes, a *flag* fica setada; portanto, deve ser apagada pelo programa;
6. O registrador SR é limpo, causando dois efeitos. Primeiro, o *bit* GIE é limpo; logo, outras interrupções mascaráveis são desabilitadas, mas interrupções não-mascaráveis permanecem ativas. Segundo, ele encerra qualquer modo de baixo consumo;
7. O conteúdo do vetor de interrupção é carregado no PC, e o programa continua com a rotina de tratamento de interrupção nesse endereço.

Resumidamente, durante uma interrupção, a CPU realiza o seguinte fluxo:

1. Armazena o endereço de retorno (PC) e o registrador SR na pilha;
2. Lê o endereço do vetor de interrupção na *Flash;*
3. Salta para a rotina associada.

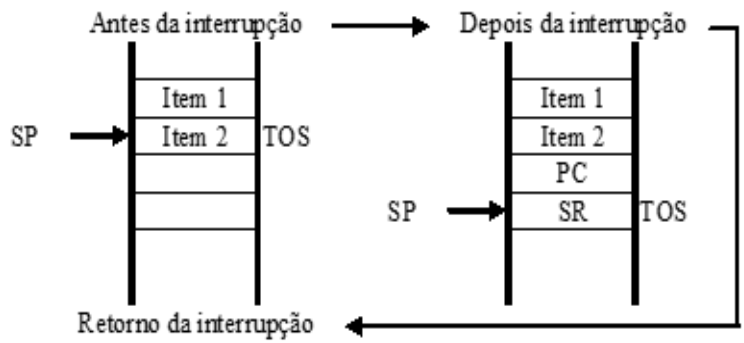
Após a execução, usa a instrução RETI para retornar, restaurando

PC e SR.

A Figura 1.1 ilustra essa sequência. Antes de desviar para a RTI da interrupção, o SP é inicialmente decrementado em dois, e o valor do PC é salvo no endereço apontado por SP. Novamente o SP é decrementado em dois para guardar, dessa vez, o valor do SR. Logo o topo da pilha (TOS) aponta para o valor de SR do programa principal antes da interrupção.

Antes de voltar para o programa principal, o valor do SR é restaurado e depois o SP é incrementado em dois. O valor do PC é restaurado, e o SP é novamente incrementado em dois. Logo, o topo da pilha (TOS) aponta novamente para o valor do item 2, voltando ao estado em que estava como era antes da interrupção.

Figura 1.1 – Sequência de operação na pilha antes, depois e no retorno da interrupção.



1.1.2 Prioridades de interrupção

As prioridades de interrupção são fixas, conforme apresentadas na tabela 1.1 (tabela de vetores de interrupção). Nessa tabela, os vetores estão em ordem decrescente de prioridade, da mais alta para a mais baixa. Caso ocorra a solicitação de mais de uma fonte de interrupção, essas prioridades determinam qual a ordem em que elas serão executadas.

1.2 Tipos de interrupções

Existem dois tipos de interrupção: as mascaráveis e as não-mascaráveis. Nas não-mascaráveis, existe um tipo especial de interrupção, a interrupção de evento de *reset*.

1.2.1 Interrupções Não-Mascaráveis (NMI)

As interrupções não-mascaráveis são tipos de interrupções que compartilham um único vetor com uma prioridade maior do que as interrupções mascaráveis. Ao contrário das interrupções mascaráveis, as interrupções NMI não podem ser desativadas pelo *bit* de habilitação de interrupção global (GIE), mas elas podem ser habilitadas por *bits* de habilitação individuais (NMIIE, ACCVIE, OFIE).

Diante a ocorrência de uma NMI, os *bits* de habilitação de interrupção NMI são automaticamente desabilitados pelo *hardware*. O próprio programa deve habilitar novamente os *bits* de interrupção NMI necessários na RTI como uma última instrução para que a interrupção seja reabilitada, se essa for a ideia. A interrupção não-mascarável possui três fontes:

- A violação de acesso à memória *flash*;
- Uma falha no oscilador;
- Uma borda ativa no pino RST/NMI quando configurado no modo NMI.

1.2.1.1 Pino reset/NMI

O pino RST/NMI é dedicado à interrupção não-mascarável, sendo a função de *reset* por padrão. O *reset* acontece quando um nível lógico baixo é aplicado no pino. A seleção entre o função de reset e NMI é configurada no registrador de controle do temporizador *watchdog*, WDTCTL.

Caso seja desejada a função NMI, o pino passa a ser uma entrada de interrupção NMI (setando o *bit* WDTNMI). Nesse caso, essa entrada gera uma interrupção caso detecte uma borda de subida ou de descida, a depender do valor do *bit* de seleção de borda WDTNMIES.

A interrupção é habilitada pelo *bit* NMIIE no registrador de função especial IE1 e a *flag* correspondente a ser setada na transição é NMIIFG no registrador IFG1.

1.2.1.2 Violação de acesso a memória Flash

A interrupção por violação de acesso à memória *flash* pode acontecer devido a algumas condições, como em uma operação de escrita no registrador FCTL2 e FCTL1 durante um ciclo de apagamento, e quando há operação de escrita quando FCTL1 e a flag WAIT=1 ou BUSY = 1. Quando acontece a violação de acesso a memória flash, a *flag* de interrupção ACCVIFG é setada. Logo, a violação de acesso à memória *flash* pode ser habilitada setando o bit ACCVIE. Com isso, a RTI pode verificar se a possível interrupção não-mascarável realmente foi causada por violação de acesso à memória flash.

1.2.1.3 Falha no oscilador

Nesse caso, a interrupção ocorre logo após a ocorrência da falha (a *flag* OFIFG é setada) do oscilador LFXT1. Após o processo da RTI ocorrer devido ao acionamento da *flag* OFIFG, ela permanece setada, mostrando como se houvesse uma falha no oscilador mesmo quando a RTI já foi tratada, mas pode ser apagada através do programa.

1.2.2 Interrupções mascaráveis

As interrupções mascaráveis, diferente das NMI, dependem de que o *bit* GIE esteja ativo para reconhecimento por parte do controle de interrupção da CPU. Cada fonte de interrupção mascarável pode ser desabilitada individualmente por um *bit* de habilitação de interrupção, ou todas podem ser desabilitadas pelo *bit* GIE no registrador de status (SR).

Se o *bit* GIE for setado (nível lógico alto) dentro de uma rotina de tratamento de interrupção, pode haver o aninhamento de interrupções. Qualquer interrupção que ocorra durante a RTI, independentemente de sua prioridade, irá interromper a rotina de tratamento de interrupção em serviço, por isso é importante que o *bit* GIE esteja limpo antes da execução da RTI, o qual é feito automaticamente pelo *hardware*.

1.3 Vetores de interrupção

Os vetores de interrupção são utilizados para a localização do código da interrupção (ou RTI), ou seja, eles apontam para a rotina de tratamento de interrupção conforme seu endereço especificado. Na Tabela 1.1, são mostrados os 13 vetores, sendo que cada vetor armazena um endereço definido na memória, além da prioridade, fonte, flag,

registrador, bit de habilitação e se é compartilhado por múltiplas flags. Abaixo, tem-se uma lista definindo os objetivos principais das colunas:

1. Prioridade – aquelas que são atendidas primeiro quando uma ou mais ocorrem ao mesmo tempo.
2. Fonte – A fonte, como o próprio nome já sugere, evidencia a fonte da interrupção, ou seja, o evento que gerou a interrupção. Como podemos ver na Tabela 1.1, cada vetor pode ter diferentes fontes de interrupção.
3. *Flag* – a *flag* nada mais é do que a sinalização da ocorrência de uma interrupção ou a solicitação da interrupção, ou seja, ela pode ser usada para indicar se uma interrupção foi gerada por parte de uma fonte de interrupção.
4. Registrador – os registradores nesse caso são responsáveis por conter as *flags* e o *bit* de habilitação da interrupção, ou seja, o *hardware* detecta a *flag* que gerou a interrupção caso o *bit* de habilitação esteja ativado.
5. Múltiplas *flags* – o vetor pode agrupar mais de uma *flag* de interrupção, por isso é importante a combinação com um *bit* de habilitação para confirmação do início do processo da RTI.

Na Tabela 1.1, os vetores de interrupção estão ordenados de forma decrescente, do vetor mais prioritário para o com menos prioridade. Essas prioridades são fixadas por *hardware*, logo, o usuário não pode alterá-las. Dessa forma, o microcontrolador é capaz de executar a RTI caso 2 ou mais sejam solicitadas, lendo o endereço do vetor da fonte de interrupção. Nos vetores de prioridade 31 e 30, temos as interrupções especiais não-mascaráveis do tipo *reset* e NMI/oscilador respectivamente. As demais interrupções de 29 a 18 são mascaráveis. O livro do módulo I desta obra contém mais detalhes sobre os vetores de interrupção.

Tabela 1.1 - Vetores e fontes de interrupção

Prioridade	Fonte	Sinalizado pela *flag*	Registrador da(o):		Habilitado pelo *bit*	Múltiplas *flags*
			flag			
31 (RESET)	POR/BOR	PORIFG	IFG1	----------	----------	sim
	Pino RST	RSTIFG				
	Watchdog	WDTIFG				
	Memória FLASH	KEYV	FCTL3			
30 (não-mascarável)	Pino NMI	NMIIFG	IFG1	IE1	NMIIE	sim
	Falha do oscilador	OFIFG			OFIE	
	Violação de acesso à memória FLASH	ACCVIFG	FCTL3		ACCVIE	
29	*TimerA1*, canal 0	CCIFG	TA1CCTL0		CCIE	não
28	*TimerA1*, canais 1 e 2	CCIFG	TA1CCTL1, TA1CCTL2		CCIE	sim
	Estouro de contagem do *timerA1*	TAIFG	TA1CTL		TAIE	
27	Comparador A	CAIFG	CACTL1		CAIE	não
26	Estouro do *watchdog* no modo temporizador	WDTIFG	IFG1	IE1	WDTIE	não
25	*TimerA0*, canal 0	CCIFG	TA0CCTL0		CCIE	não
24	*TimerA0*, canais 1 e 2	CCIFG	TA0CCTL1, TA0CCTL2		CCIE	sim
	Estouro de contagem do *timerA0*	TAIFG	TA0CTL		TAIE	
23	USCI_A0/USCI_B0 recepção USCI_B0 I^2C status	UCA0RXIFG, UCB0RXIFG	IFG2	IE2	UCA0RXIE, UCB0RXIE	sim
22	USCI_A0/USCI_B0 transmissão USCI_B0 I^2C recepção/transmissão	UCA0TXIFG, UCB0TXIFG	IFG2	IE2	UCA0TXIE, UCB0TXIE	sim
21	ADC10 – fim de conversão	ADC10IFG	ADC10CTL0		ADC10IE	não
19	Porta 2 - mudança de estado do pino	P2IFG.0 a P2IFG.7	P2IFG	P2IE	Bit 0 a 7	sim
18	Porta 1 - mudança de estado do pino	P1IFG.0 a P1IFG.7	P1IFG	P1IE	Bit 0 a 7	sim

1.4 Registradores

Os registradores, dentro do escopo de interrupções, são utilizados para fazer o controle dessas interrupções. Cada registrador pode armazenar *flags* de interrupção ou *bits* de habilitação de interrupção, e possui seu próprio endereço. Dessa forma, podem ser acessados pelo usuário a fim de habilitar ou desabilitar essas interrupções, sendo eles denominados registradores de controle de interrupção.

Os registradores de controle de habilitação de interrupção são:

- IE1: da interrupção NMI e do *watchdog*;
- IE2: da USCI_A e da USCI_B;
- TA1CCTL0, TA1CCTL1, TA1CCTL2: canais 0 a 2 do *timerA1;*
- TA0CCTL0, TA0CCTL1, TA0CCTL2: canais 0 a 2 do *timerA0;*
- TA1CTL: estouro do *timerA1;*
- TA0CTL: estouro do *timerA0;*
- P1IE, P2IE: portas 1 e 2;

- ADC10CTL0: ADC10;
- CACTL1: comparador analógico.

Já os registradores que indicam o evento de interrupção são:

- IFG1: da interrupção NMI e do *watchdog* e de *reset*;
- IFG2: da USCI_A e da USCI_B;
- TA1CCTL0, TA1CCTL1, TA1CCTL2: canais 0 a 2 do *timerA1;*
- TA0CCTL0, TA0CCTL1, TA0CCTL2: canais 0 a 2 do *timerA0;*
- TA1CTL: do estouro do *timerA1;*
- TA0CTL: do estouro do *timerA0;*
- P1IE, P2IE: portas 1 e 2;
- ADC10CTL0: ADC10;
- CACTL1: comparador analógico.

Por fim, um registrador especial interno da CPU, o SR, possui o *bit* de habilitação global de interrupções GIE. Esse *bit* pode ser usado pelo usuário para habilitar ou desabilitar todas as interrupções, menos as não-mascaráveis, já que elas não podem ser desabilitadas pelo programa. O registrador não é acessível ao usuário, por isso não tem um endereço.

1.5 Modos de baixo consumo

O LPM, ou *Low power mode*, é um mecanismo para economizar energia ou, de certa forma, aumentar a eficiência energética do microcontrolador. Afinal, é importante que o microcontrolador tenha características como respostas em alta velocidade de processamento e um baixo consumo de energia. Desde os seus primeiros passos, foi projetado para atuar em modo de baixo consumo de energia (*Low Power Modes*), e isso reflete em uma gama de possibilidades em projetos que demandam o baixo consumo de energia.

Existem 5 modos de operação em baixo consumo, porém dois deles (modos 1 e 2) raramente trabalham em dispositivos atuais (Eram úteis em versões anteriores do DCO). Os mais importantes serão sintetizados aqui.

Em relação ao modo ativo e, com destaque, aos LPMs 0, 3 e 4, tem-se:

Modo ativo (MA): CPU, todos os *clocks* e módulos estão ativos, com uma corrente de I = 0,3 mA. O microcontrolador inicializa neste modo, que deve ser utilizado quando a CPU for necessária.

***Low Power Mode* 0 (LPM0):** Este modo pode ser utilizado

quando a CPU não é requerida, e os módulos requerem um *clock* rápido de SMCLK e o DCO. Além disso, a CPU e o MCLK permanecem inativos, e o SMCLK e o ACLK permanecem ativos. A corrente de operação neste modo é de 0,085 mA. O *bit* CPUOFF desativa a CPU.

***Low Power Mode* 3 (LPM3):** Este modo é padrão de baixo consumo quando o dispositivo deve "acordar" em intervalos regulares e precisa de um estágio de relógio lento. A CPU, MCLK, SMCLK e DCO estão todos desabilitados. Apenas o ACLK permanece em modo ativo. A corrente de operação é de 0,001mA. A corrente pode ser reduzida pela metade usando o VLO ao contrário de um cristal externo nos pinos P2.6 e P2.7.

***Low Power Mode* 4 (LPM4):** A CPU e todos os *clocks* estão desabilitados. A corrente de operação neste modo é aproximadamente de 0,0001 mA, uma corrente bem baixa de operação. O dispositivo só pode ser ativado por um sinal externo. Isso também é chamado de modo de retenção de RAM.

1.6 Exemplos de utilização

Abaixo o *timer0* é configurado para contar até o valor de 10000 (valor carregado em TA0CCR0), e quando chega lá, ele aciona uma

interrupção, desviando o fluxo de execução para a RTI do timer. Na RTI do timer, o led muda de estado (liga ou desliga). Repare que o código principal fica em laço infinito vazio, enquanto a RTI do timer cuida de comutar o estado do *led*. Considerando o SMCLK em 1 MHz, o *led* irá comutar de estado (ligar ou desligar) a cada 0,08 segundos.

```
#include <msp430.h>

void main(void) {
WDTCTL = WDTPW | WDTHOLD; // Desabilita o
watchdog timer
P1DIR |= BIT0;  //Configura P1.0 como saída
(LED)
P1OUT &= ~BIT0; //Inicializa P1.0 desligado
TA0CCTL0 = CCIE;//Habilita interrupção no canal
0 do Timer_A
TA0CCR0 = 10000;//Valor de comparação, definindo
o período
TA0CTL = MC_1 + TASSEL_2 ID_3;// modo up,
Seleciona SMCLK/8
//modo up (contagem   até TA0CCR0)
```

```
__bis_SR_register(GIE);    //Habilita
interrupções globais
//Loop principal vazio, pois tudo ocorre na
interrupção
while (1){}
}

// Vetor de interrupção do canal 0 do Timer_A
#pragma vector = TIMER0_A0_VECTOR
__interrupt void Timer_A(void) {
   P1OUT ^= BIT0;// Alterna o estado do pino
   P1.0 (liga/desliga led)
}
```

No exemplo a seguir, o funcionamento é o mesmo, mas com a vantagem de reduzir drasticamente o consumo por meio do uso do modo de baixo consumo LPM0.

```
#include <msp430.h>
//código com a versão com modo de economia de
energia LPM0
void main(void) {
```

```
WDTCTL = WDTPW | WDTHOLD;  // Desabilita o
watchdog timer

P1DIR |= BIT0;// Configura P1.0 como   saída
(LED)
P1OUT &= ~BIT0;            // Inicializa P1.0
desligado
TA0CCTL0 = CCIE; // Habilita interrupção
no canal 0
// do Timer_A
TA0CCR0 = 10000;// Valor de comparação para
o timer (define o período)
TA0CTL = MC_1 + TASSEL_2 ID_3;// modo up,
Seleciona SMCLK/8
//modo up (contagem até TA0CCR0)
__enable_interrup();// habilita GIE
for (;;) {
  __low_power_mode_0();// entra no modo LPM0
}
}
```

```
// Vetor de interrupção do canal 0 do Timer_A
// o processador retorna automaticamente para
o     modo LPM0 ao retonar
#pragma vector = TIMER0_A0_VECTOR
__interrupt void Timer_A(void) {
P1OUT ^= BIT0;// Alterna o estado do pino P1.0
(liga/desliga led)
}
```

Capítulo 2 – TIMERS E O WATCHDOG TIMER

2.1 Timers A

O uso de temporizadores (*timers*) e do *Watchdog Timer* (*WDT*) em sistemas de microcontroladores é essencial para o desenvolvimento de aplicações que requerem controle preciso do tempo e maior confiabilidade. Enquanto os *timers* podem gerar temporizações, o *watchdog timer* tem a função primária de resetar o microcontrolador caso alguma falha ocorra.

O MSP430G2553 possui duas instâncias de *timer* A: o *timer* A0 e o *timer* A1. Abordaremos neste capítulo, o funcionamento desses *timers* e do *WDT*, diagramas de blocos lógicos, alguns de seus registradores e exemplos práticos de sua aplicação.

O *timer* A é um temporizador ou contador de 16 *bits* que tem várias funções importantes. Ele oferece três canais para realização de captura, comparação ou PWM. O *timer* pode funcionar com o *clock* externo ou interno.

Convém dizer que os dois *timers* A podem ser utilizados no modo

contador, caso a funcionalidade de temporização não seja selecionada para eles.

2.1.1 Características dos timers A

Os *timers* A tem as seguintes características:

- quatro modos de operação;
- A fonte de *clock* poderá ser configurada tanto para externa (TACLK) quanto interna (ACLK, SMCLK);
- contadores de 16 *bits*;
- podem gerar uma interrupção no estouro da contagem;
- Base de tempo para a captura, comparação e PWM;
- Seus três canais podem realizar captura assíncrona ou síncrona de entrada;
- No caso do *timer* A0, pode iniciar uma conversão A/D através da saída de comparação do canal 0 ou 1 ou 2 e uma captura pode ser realizada na saída do comparador A+;

2.1.2 Canais dos timers A

Cada timer A do MSP430G2553 tem três canais que podem ser configurados independentemente e seletivamente no modo de captura ou comparação ou PWM.

O canal 0 do *timer* A0, por exemplo, é configurado através do registrador TA0CCTL0, o canal 1 por meio do registrador TA0CCTL1, e o canal 2 por meio de TA0CCTL2. Ou seja, via de regra, o canal x do *timer* Ay é configurado por meio de TAyCCTLx, sendo y igual 0 ou 1 conforme o *timer* escolhido, e x sendo 0, 1 ou 2 para o canal escolhido.

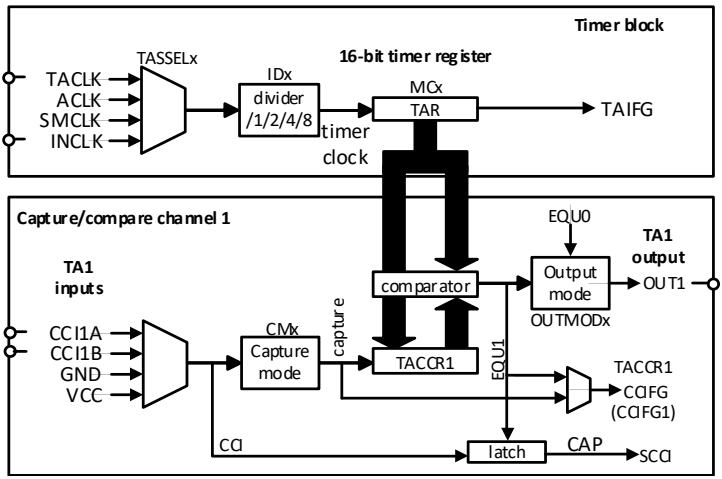
O canal 1 do *Timer* A1, por exemplo, é configurado por meio do registrador TA1CCTL1, que controla o modo de comparação/captura ou PWM. Esse canal pode operar tanto no modo captura quanto no modo comparação ou PWM, dependendo da aplicação.

De forma geral, os canais do *timer* A0 tem conexões com sinais internos como CAOUT (saída do comparador A+) e o *clock* ACLK, além de conexões com sinais externos em alguns pinos. Uma saída de qualquer canal desse *timer* pode ser utilizada para iniciar uma conversão A/D. Já os canais do t*imer* A1 possui somente conexões nos pinos, sendo mais utilizado para se conectar a dispositivos externos.

Na Figura 2.1, temos o diagrama em blocos da estrutura do *timer*

A que fornece uma visão mais geral e detalhada desse *timer*, mostrando a interação entre o registrador contador (TAR), os canais CCRx, o *clock* de entrada e os sinais de controle. Esse diagrama é essencial para compreender o funcionamento interno do *Timer* A e os três canais de captura/comparação.

Figura 2.1 - Diagrama de bloco do timer A



Cada canal pode utilizar um pino específico. Vale ressaltar que para usar corretamente os modos de comparação e captura/ PWM, é necessário configurar os pinos para que assumam a função correspondente ao modo selecionado.

As tabelas a seguir apresentam os canais do *timer* A0 e *timer* A1 e seus respectivos pinos.

Tabela 2.1 - Canais do timerA0 e seus pinos

***Timer* A0**	**Pinos de comparação e PWM**	**Pinos de comparação/PWM/captura Entrada CCIA**
Canal 0	1.5, 3.4	1.1
Canal 1	3.5, 1.6, 2.6	1.2
Canal 2	3.6	3.0

Tabela 2.2 - Canais do timerA1 e seus pinos

***Timer* A1**	**Pinos de comparação e PWM**	**Pinos de comparação/PWM/captura (Entradas: CCIA, CCIB)**
Canal 0	3.1, 2.3	2.0, 2.3
Canal 1	2.1, 3.2	2.1, 2.2
Canal 2	3.3, 2.4	2.4, 2.5

2.1.3 Modo de Operação

O contador do timer de 16 bits é um registrador chamado de TAR que incrementa na borda de subida do *clock* selecionado. O timer pode gerar interrupção sempre que acontecer *overflow* do registrador TAR. *Através* do seletor de fonte de *clock* do timer pode ser selecionado o *clock* ACLK, SMCLK, TACLK ou INCLK, sendo que ACLK E SMCLK são fontes internas e TACLK E INCLK são externas. Com a fonte de c*lock* selecionada e através dos *bits* IDX, a frequência da fonte pode ser dividida para o *timer* por um fator de 2, 4 ou 8. Caso a frequência não seja necessária ser dividida, basta escolher o valor 0 para os *bits* IDX.

O *timer* pode ser iniciado quando os *bits* do MCx forem maiores que zero e a fonte de *clock* estiver ativa. Nos modos *up* e *up/down* podemos zerar a contagem do *timer* colocando TAyCCR0 igual a zero. Para iniciar a contagem, basta atribuir um valor diferente de zero no registrador citado que o *timer* começará a contar a partir do zero.

O *timer* tem quatro modos de operação que podem ser selecionados nos bits do MCx, que são o (00) modo *stop*, (01) *up*, (10)

contínuo e (11) *up/down*, que serão explicados cada modo a seguir:

- **modo *stop***: a contagem do TAR permanecerá parada, não podendo ser incrementada ou decrementada;
- **modo *up***: o contador TAR será incrementado a cada pulso do *clock* até atingir o valor que foi configurado no TAxCCR0. Quando TAR alcança esse valor, a *flag* do canal é setada para sinalizar que o valor do TAR alcançou o valor do TAxCCR0. Nesse caso, no próximo pulso do *clock* o *timer* reiniciará sua contagem a partir do zero;
- **modo contínuo**: nesse modo, o contador TAR contará de 0 a 65.535 (valor máximo). Após chegar no máximo, a contagem é reiniciada para zero no próximo pulso do *clock*, e a *flag* TAIG vai para 1;
- **modo *up/down*:** o contador vai contar até o valor configurado no TACCR0. Após atingir esse valor, o contador será invertido, fazendo com que a contagem seja decrescente, ou seja, será do valor configurado até zero. De outro modo, o contador ficará invertendo a de

zero ao valor de TACCR0 e do valor até zero.

2.1.4 Registradores

Os registradores do t*imer* A armazenam informações para controlar e verificar o funcionamento do temporizador. Eles são essenciais na configuração do *timer* A. São eles: TAyCTL, TARy, TAyCCTLX, TAyCCRX.

TAyCTL é o registrador de controle do *timer* A, responsável em configurar e selecionar a fonte de *clock* que vai ser usada nesse *timer*, podendo ser: o TACLK, que é o *clock* externo; SMCLK, o *clock* secundário; ou ACLK, o *clock* auxiliar. É possível ajustar o divisor de *clock*, que pode ser divido em 2, 4 ou 8 antes de chegar no TAR, o que é feito através dos *bits* IDX.

TARy é um registrador de contagem de 16 *bits*, no qual ele incrementa e decrementa, dependendo do modo de contagem que pode ser configurado no *timer* A.

TAyCCTLX é um registrador que configura cada canal do *timer* A, determinando se o canal será configurado no modo de captura, comparação ou PWM. Nesse registrador, é possível também habilitar a

interrupção do canal.

TAyCCRX é o registrador em que é armazenado o valor de comparação ou captura.

2.1.5 Reset de Contadores

O contador de 16 bits TAR pode ser resetado de três formas diferentes. A primeira é escrevendo diretamente no registrador; após ser resetado, ele começará a contar do zero. A segunda forma é escrever o valor zero no TAyCCR0, mas isso funcionará apenas se estiver no modo contínuo. A terceira forma é setar o *bit* TACLR no registrador TAyCTL. Esse *bit* é responsável por resetar a contagem, ou seja, ele coloca o TARy em zero.

2.2 Modo dos canais

2.2.1 Modo de captura

O modo de captura é usado para medir tempos, ou seja, serve para capturar determinados eventos no tempo, que pode ser a largura

de um pulso ou período de um sinal, podendo gerar interrupções no momento exato da captura de forma eficaz. O canal, nesse modo, pode funcionar de forma assíncrona ou síncrona com o *clock*.

Por meio do registrador TAyCCTLx, é possível configurar a detecção de borda para subida, descida ou ambas. Com a seleção da borda desejada, no momento em que ela ocorrer, o valor do *timer* é instantaneamente copiado para o registrador TAyCCRx, com precisão e sem intervenção da CPU. Nesse momento, a *flag* CCIFG do canal é ativada, indicando que a captura foi realizada com sucesso. A partir da diferença entre os valores de TAyCCRx em duas detecções de borda de subida, pode-se determinar, por exemplo, o período de um sinal de alta frequência com alta precisão.

2.2.2 Modo de Comparação/PWM

O modo de comparação é utilizado para gerar eventos em momentos específicos do temporizador, como gerar interrupções periódicas ou gerar sinais PWM.

No modo de comparação, o valor de TARy é comparado com o valor definido em TAyCCRx. Quando ocorre a igualdade, o timer pode

gerar uma interrupção ou controlar a saída.

O modo de comparação e PWM do *timer* pode ser utilizado para geração de interrupções periódicas precisas e na geração de sinal PWM. Esses eventos ocorrem quando o registrador de contagem TARy atinge o valor de comparação armazenado em TACCRX, ativando o sinal interno EQUx. Esse evento de igualdade é utilizado para controlar a saída de comparação, conforme definido pelos *bits* OUTMODX no registrador TAyCCTLx.

2.3 Interrupções do Timer A

O *timer* A possui múltiplas fontes de interrupções, essenciais para a execução de tarefas com precisão temporal. Existem quatro fontes de interrupção distintas: interrupção do canal 0 (TACCR0:CCIFG); estouro da contagem do *timer* (TAIFG); e interrupções dos demais canais CCP (TACCR1:CCIFG até TACCR2:CCIFG).

O *timer* A possui dois vetores de interrupção distintos: um dedicado exclusivamente à interrupção do Canal 0 (TAxCCR0), e um segundo vetor para as interrupções dos demais canais (TAyCCRx) e para o estouro do *Timer* (TAxIFG). Quando uma interrupção ocorre e está habilitada no Canal 0, o fluxo do programa é desviado para o

vetor correspondente, e o *flag* CCIFG é automaticamente apagado pelo *hardware*. Para os demais canais e estouro de contagem do *timer*, ocorrendo qualquer ativação da *flag* dos canais 1, 2, ou ainda da *flag* TAIFG, o PC é direcionado para um único vetor de interrupção. Através do registrador TAIV, é possível determinar de forma eficiente quem provocou o desvio para esse vetor. TAIV assume o valor 2, 4 e 10, quando a fonte que gerou a interrupção for o canal 1, 2 e estouro de contagem do timer respectivamente.

2.4 Exemplos de Utilização

O código a seguir configura o canal 0 no modo de comparação do *timer* A0 para gerar interrupções a cada 50.000 ciclos de *clock* do SMCLK. Quando ocorrer a interrupção, o estado do pino P1.0 é alternado, fazendo com que um LED conectado a esse pino pisque, alternando o estado do LED a cada 50 milisegundos.

```
#include <msp430.h>

void main(void) {
```

```
WDTCTL = WDTPW | WDTHOLD; // Desabilita o
watchdog timer
P1DIR |= BIT0;   // Configura P1.0 como saída
(LED)
P1OUT &= ~BIT0;  // Inicializa P1.0 desligado
TA0CCTL0 = CCIE; // Habilita interrupção no
canal 0 do Timer_A
TA0CCR0 = 50000; //Valor de comparação, definindo
o período
TA0CTL = TASSEL_2 + MC_1;//Seleciona SMCLK como
clock e modo //up contagem até TA0CCR0)
__bis_SR_register(GIE); // Habilita interrupções
globais
while (1) {// Loop vazio, pois tudo ocorre na
interrupção
}
}

// Vetor de interrupção do canal 0 do Timer_A
#pragma vector = TIMER0_A0_VECTOR
```

```
___interrupt void Timer_A(void) {
P1OUT ^= BIT0;//Alterna o estado do pino
P1.0(liga/desliga LED)
TA0CCR0 += 50000; //atualizar o valor de
comparação
}
```

Esse código a seguir configura o microcontrolador para comutar um LED no P1.0, usando o *timer* A1 em modo contínuo com interrupção no estouro da contagem. Com a fonte de *clock* SMCLK operando em 1 MHz e dividida por 8, o *timer* vai levar 0,524 segundos para contar de 0 até 65535 e voltar para 0. A cada estouro de contagem do *timer* nesse intervalo de tempo, o LED terá seu estado comutado. Observe que o *loop* do código está vazio, pois todo o controle é feito por interrupções.

```
#include <msp430.h>

void main(void) {
WDTCTL = WDTPW | WDTHOLD;
P1DIR |= BIT0; //configura o p1.0 como saída
```

```
P1OUT &= ~ BIT0; // zera a saída em nível baixo
TA1CTL = TASSEL_2 | MC_2 | ID_3 | TAIE;
// TA1CTL configura o timer A1
// TASSEL_2 smlck em 1MHz
// MC_2 modo contínuo
//ID_3 divide o clock por 8 onde fica 125.000 Hz
// TAIE habilita interrupção no estouro da
contagem
while(1) { }
}

#pragma vector = TIMER1_A1_VECTOR //vetor de
interrupção do timer A1
__interrupt void Timer1_A1_ISR (void) {
switch (TA1IV) {// TA1IV indica qual evento
gerou interrupção
      case TA1IV_TAIFG:// indica o estouro de
      contagem do timer
      P1OUT ^= BIT0; // inverte o estado do led
      break;
```

```
}
}
```

Por fim, mostra-se a brilhante utilização dos três canais no modo de captura do Timer A0 para medir o tempo entre as bordas de subida dos três canais, os quais são conectados a um sistema trifásico de 60 Hz, sendo um canal para cada fase. Sequencialmente, determina-se o tempo entre as bordas capturadas pelos canais. Caso esse tempo fique inferior a 5,09 ms, LEDs são acionados para indicar essa condição. Isso tem como aplicação sinalizar uma falha na rede elétrica, como um desequilíbrio de fase.

```
#include <msp430.h>
#include <stdint.h>
#include <stdlib.h>

//******************defines***************
//a) parametros
#define TRUE 1
#define FALSE 0
#define A 1
```

```
#define B 2
#define c 3//c minusculo pois com C maiusculo
ele já está pre-definido

//b) BITS da P2 do bit 1 ao bit 7
#define MEDIDOR_ROTACAO BIT1
#define LED_FALTA_BIT_0 BIT2
#define LED_FALTA_BIT_1 BIT3
#define LED_FALTA_UMA_FASE BIT4
#define LED_FALTA_FASES BIT5
#define LED_SEQ BIT6
#define LED_TIMEOUT_ROTACAO BIT7

//c) tempos
#define ANG_T_INF 5092
#define T180 8333
#define ANG_T_SUP 6018
#define TEMPO_MAXIMO_MEDICAO_PERIODO 38709
#define TEMPO_MINIMO_MEDICAO_PERIODO 29268
```

```
//**********Protótipo de funções***********
void encontre_seq();
void conf_pinos();
void enable_medicao();
void disable_medicao();

//**********variáveis globais**************
uint8_t calc=0, seq=1, faltaDesb=0, par, pos,
angulo_normal[4], faseA_wdt=0, faseB_wdt=0,
faseC_wdt=0, Falta2ou3Fases = 0, Falta1Fase =
0;

int8_t var_l=1, v[2]={3,1};
uint8_t etapa=0, instante_pwm=1, feito = 0, i =
0, espera1Borda = 0;
uint16_t angt, cap_Ant, ti = 0, periodo_volta,
tick = 0, ti_pwm, resultado, rotacao, tac_sw =
8000, var = 29000/2;

void main(void){
```

```
WDTCTL = WDTPW | WDTHOLD;//Stop watchdog timer
conf_pinos();

//configurações de clock para 1 MHz
DCOCTL = 40;//calibrado para 1 MHz a depender
de cada MSP
BCSCTL1 = 135;

//configurar os canais do timer0 e o proprio
timer0
TA0CTL = TASSEL_2 + MC_2 + TACLR + TAIE;//
TA0CCTL0 = CM_1 + CAP + CCIE;//captura com
interrupção
TA0CCTL1 = CM_1 + CAP + CCIE;
TA0CCTL2 = CM_1 + CAP + CCIE;//todos para a
borda de subida

//conf. timer1
TA1CTL = TASSEL_2 + MC_2 + TACLR;//
```

```
__bis_SR_register(GIE);//habilita as
interrupções

while(1){//LPM0;
if (calc){ calc = 0;
      var_l = v[1] - v[0];
      encontre_seq();//com base na var_l, no
     angt e
      v[0] = v[1];//desloca posição (shift data)
                        if ( angt < ANG_T_INF )
                  {//o  angulo  que  estiver  //
                  menor que 120 graus, ou seja,
                  tem problema?
                  if (seq){//diz qual é a falta
                        faltaDesb = v[1];
                  } else {//sequencia negativa
                        faltaDesb = v[1] + 1;
                  if (faltaDesb > 3) faltaDesb =
1;
                  }
```

```
                angulo_normal[v[1]] = FALSE;

            } else { //angulo não está
        abaixo do limite //inferior,
        portanto não há faltaDesb de
        fase
                    angulo_normal[v[1]]
            = (angt < ANG_T_SUP)?
            TRUE: FALSE;    uint8_t
            fase;
                    for (fase = 3; fase
            && angulo_normal[fase];
            fase--);

                    if (!fase){//todos
            os tres angulos estao em
            120 graus?
             faltaDesb = 0;
             Falta1Fase = 0;
        }
```

```
            }
            //externar falta, e seq para três leds e
            falta de fase
            seq? (P2OUT |= LED_SEQ) : (P2OUT &= ~LED_
             SEQ);
            Falta1Fase?  (P2OUT  |=  LED_FALTA_UMA_
FASE)       : (P2OUT &= ~LED_FALTA_UMA_FASE);
            Falta2ou3Fases?  (P2OUT  |=  LED_FALTA_
FASES) : (P2OUT &= ~LED_FALTA_FASES);

            switch (faltaDesb) {
            case A:
                  P2OUT &= ~LED_FALTA_BIT_1; //01
                  P2OUT |= LED_FALTA_BIT_0;
                  break;
            case B:
                  P2OUT |= LED_FALTA_BIT_1; //10
                  P2OUT &= ~LED_FALTA_BIT_0;
                  break;
            case c:
```

```
                    P2OUT |= (LED_FALTA_BIT_1
                | LED_FALTA_BIT_0); //11
            break;
        case 0:
                    P2OUT   &=   ~(LED_FALTA_
                BIT_1 | LED_FALTA_BIT_0); //00
            break;
        } //fim do swtich
        //bloco de externação de variaveis para
        //leds
}//fim do if calc

static char primeira_vez = 1;
if (!etapa){
        if (primeira_vez){//acao de chegada
            primeira_vez = 0;
            disable_medicao();
            feito = 0;
            ti = tick;
        }
```

```
            if ( (tick - ti) >= 1){// a
         cada 65 //milisegundo
        espera1Borda = 0;
        enable_medicao();
        etapa = 1;
}
} else if (feito >= 1){ //se em etapa = 1
e feito
if (feito == 1){
        //trata da saturação
        if  (periodo_volta  <  TEMPO_MINIMO_
MEDICAO_PERIODO){
        rotacao = 2050;
}
else {
        rotacao = (uint32_t)(60000000)/
        periodo_volta;//
}

            } else if (feito == 2){//deu
```

```
            timeout-> logo passou //do tempo
            maximo
            rotacao = 1550;//rotacao é 1.550
            rpm,
      }
            if (rotacao <= 1550) rotacao
            = 1550;//dominio da equação. se for
            abaixo, tac_sw ocorre underflow
      tac_sw =(rotacao*16 - 24800)<<1;

      etapa = 0;
      primeira_vez = 1;
}//else feito >= 1

//PWM por software
if (instante_pwm == 1){//setar
      if ( (TA1R - ti_pwm) >= 16000){
            P3OUT |= BIT7;
            ti_pwm = TA1R;
            instante_pwm = 0;
```

```
      }
} else {//zerar
      if ( (TA1R - ti_pwm) >= tac_sw){
            P3OUT &= ~BIT7;
            instante_pwm = 1;
      }
}

}//fim do laço infinito
}//fim da main


#pragma vector=TIMER0_A0_VECTOR//canal 0
__interrupt void Timer0_canal_0(void) {
if (calc)  Falta1Fase = 1;
v[1] = A;
angt = CCR0 - cap_Ant;
cap_Ant = CCR0;
faseA_wdt = 0;
calc = 1;
```

```
}//fim da RTI do canal 0


#pragma vector=TIMER0_A1_VECTOR //canais 1 e 2
overflow
__interrupt void Timer0_canais1_2_ovf(void) {
switch (TA0IV) {
case TA0IV_TACCR1:
if (calc)   Falta1Fase = 1;
v[1] = B;
angt = CCR1 - cap_Ant;
cap_Ant = CCR1;
faseB_wdt = 0;
calc = 1;
break;
case TA0IV_TACCR2:
if (calc)   Falta1Fase = 1;
v[1] = c;
angt = CCR2 - cap_Ant;
cap_Ant = CCR2;
```

```
faseC_wdt = 0;
calc = 1;
break;
case TA0IV_TAIFG:
          if ( (++faseA_wdt > 20) || (++faseB_
     wdt > 20) || (++faseC_wdt > 20) ){ //mais
     de 20 estouros do timer0
     Falta2ou3Fases = 1;
     P2OUT |= LED_FALTA_FASES; //ausencia de
     fases
}
else
     Falta2ou3Fases = 0;
tick++;//timer0 como base de tempo de 65,5 ms
break;
}//fim do switch
}//fim da RTI

void encontre_seq(){
pos = (var_1 > 0)? 1: 0; //positivo ou negativo?
```

```
par = abs(var_1 % 2) ^ 1;
seq = pos ^ par;
}

void conf_pinos(){
//configurar pinos P1.1 e P1.2 como entrada de
captura
//p1.1 canal 0, //p1.2 canal 1, //p3.0 canal 2,
A B C respectivamente
//portas p1.1 e p1.2
P1DIR &= ~(BIT1 + BIT2);    P1SEL  |=  BIT1  +
BIT2; P1SEL2 &= ~(BIT1 + BIT2);//porta p3.0
P3DIR &= ~BIT0;  P3SEL |= BIT0;   P3SEL2 &=
~BIT0;
P3DIR |= BIT7;//pwm por software para o medidor

P2SEL &= ~(BIT6 + BIT7), P2SEL2 &= ~(BIT6 +
BIT7);
P2REN |= (LED_SEQ | LED_FALTA_BIT_1 | LED_FALTA_
BIT_0 | LED_FALTA_UMA_FASE | LED_FALTA_FASES
```

```
| LED_TIMEOUT_ROTACAO);
     P2DIR |= (LED_SEQ | LED_FALTA_BIT_1 |
LED_FALTA_BIT_0 | LED_FALTA_UMA_FASE | LED_
FALTA_FASES | LED_TIMEOUT_ROTACAO);//Leds com
resistores já //em série

// 2. Atribuir a função do Timer (TA1.1) ao P2.1
// O MSP430G2553 mapeia TA1.1 (Timer A1, Canal
1) para P2.1.
P2DIR &= ~MEDIDOR_ROTACAO; P2SEL |= MEDIDOR_
ROTACAO;   P2SEL2 &= ~MEDIDOR_ROTACAO;

//ativando os resistores de pullup nos pinos
de   captura do timer0
P1REN |= BIT1 + BIT2;//resitor de
P1OUT |= BIT1 + BIT2;//pullup
P3REN |= BIT0;//resitor de
P3OUT |= BIT0;//pullup
}
```

```
void enable_medicao(){
      TA1CCTL1 = CM_1 + CAP + CCIE;//borda de
      subida, captura com //interrupção, P2.1
      no pino 10
}

void disable_medicao(){
      TA1CCTL1 = CM_0 + CAP + CCIE;//borda de
      subida, captura com //interrupção, P2.1-
      10
      TA1CCTL2 = 0;//modo de comparação, canal
      2 como espécie de wdt //para tempo máximo
      de medição de periodo
}
```

2.5 Watchdog timer

Temporizador presente em todos os microcontroladores MPS430, que atua como um "cão de caça" vigiando o funcionamento do *software* e, caso identifique uma falha, reinicia o sistema. Consiste

em contador de 16 bits que, ao fim de um dado intervalo de contagem escolhido, poderá ocasionar um *reset* da CPU. Mas pode funcionar como um simples temporizador sem gerar sinal de *reset* no estouro de sua contagem.

2.5.1 Modo Watchdog

Aplicação essencial para assegurar a confiabilidade e a segurança do sistema, ele previne as falhas e os travamentos. Nesse modo, o programador precisa frequentemente apagar a contagem do contador para ser evitado o estouro, setando o *bit* WDTCNTCL no registrador WDTCTL, com o objetivo de indicar que o *software* principal está sendo realizado sem erros e sem travamentos. Se por acaso o contador não for resetado dentro do tempo que foi programado e atingir o estouro, ele expira. Isso seta o *flag* WDTIFG situado no registrador IFG1 e dispara, especificamente, um *reset*, conhecido como PUC (*Power Up Clear*). O reset PUC também pode ocorrer em função da violação da senha de acesso ao registrador de controle. O WDT+ possui um recurso para assegurar que sua fonte de *clock* não seja desabilitada; falhando a fonte de *clock* (ACLK ou SMCLK) o *clock* do WDT+ muda instantaneamente para MCLK ou, caso for preciso, ativa o DCO. Porém, o microcontrolador

é impedido de entrar em modos de baixo consumo caso precisa dessa fonte do *clock*.

2.5.2 Modo Temporizador

Diferente do Modo *Watchdog*, que atua como um meio de segurança, resetando o sistema em caso de falha ou travamento no sistema, nesse modo ocorre interrupções regularmente em períodos programados, ou seja, ele atua como um temporizador periódico. Ao invés de ser gerado um *reset*, como no modo anterior, a expiração do *timer* ocorre quando o contador atinge o seu estouro. A escolha deste modo é realizada setando o *bit* WDTTMSEL para 1 no registrador protegido WDTCTL. No estouro da contagem, seta o *flag* WDTIFG que, se os *bits* WDTIE e GIE estiverem setados, redireciona a execução para o vetor de interrupção de número 10. Esse modo não possui ferramentas de *fail-safe* para a fonte de *clock*, ou seja, isso significa a confiabilidade da geração de interrupções periódicas pelo WDT+ neste modo depende diretamente da estabilidade e disponibilidade contínua da fonte de *clock* selecionada, sem as proteções automáticas que são ofertadas apenas pelo Modo *Watchdog*.

2.5.3 Registradores do Watchdog

→ **WDTCTL:** Registrador principal que controla todos os usos do *Watchdog*, é protegido por senha. Para a escrita dele, deve sempre ser inserido o valor 0x5A no conteúdo do *byte* mais significativo. Se um valor diferente é escrito nesse *byte* durante a tentativa da escrita, o restante do conteúdo não será modificado e acontecerá um *reset* PUC, independentemente do seu modo de operação. Para a sua leitura, o valor que retornará ao *byte* superior será o 0x69. Esse registrador possui os seguintes *bits* de controle: WDTPW (*bits* 15-8), senha de acesso; WDTHOLD (*bit* 7), parada do timer quando este *bit* está ativado; WDTNMIES (*bit* 6), seleciona a borda de interrupção NMI quando WDTNMI está ativado; WDTNMI (*bit* 5), seleciona a função do pino RST/NMI, reset = 0 e NMI = 1; WDTTMSEL (*bit* 4), seleciona o modo de operação do WDT, 0 = Modo Watchdog e 1 = Modo Temporizador; WDTCNTCL (*bit* 3), apaga a contagem do *Watchdog*; WDTSSEL (*bit* 2), seleciona a fonte de *clock*, 0 = SMCLK e 1 = ACLK; WDTISx (*bits* 1-0), seleciona o intervalo de tempo para a ocorrência do estouro ou da interrupção.

→ **IE1:** Registrador que possui *bits* que habilitam e desabilitam as interrupções de funções especificas, sendo o *bit* WDTIE o mais

relevante. Este *bit* habilita a interrupção por estouro quando ele estiver operando no Modo Temporizador e não é preciso no Modo *Watchdog*.

→ **IFG1:** Registrador que sinaliza as interrupções, possuindo *flags* que apontam a circunstância dos eventos de suspensões específicas, sendo o *flag* WDTIFG o mais relevante. Essa *flag* é setada quando ocorre um estouro; no Modo Temporizador, ela é resetada automaticamente quando a solicitação da interrupção é atendida ou é resetada por *software*. No outro modo, essa *flag* setada indica que o *Watchdog* causou um *reset* por estouro ou por violação da segurança.

2.5.4 Exemplo de Utilização

Neste exemplo, o WDT está sendo configurado como um temporizador, gerando uma interrupção periódica e alternando a cada 1 segundo o estado dos LEDs, configurados como saídas nos pinos P1.0 e P1.6.

```
#include <msp430.h>

void main(void) {
WDTCTL = WDTPW | WDTHOLD;//Desabilita o watchdog
inicialmente
```

```
P1DIR |= BIT0 | BIT6;  // Define P1.0 e P1.6
como saída
P1OUT &= ~(BIT0 | BIT6); // Inicializa os LEDs
apagados
P1REN |= BIT0 | BIT6; // Ativa resistor de
pull-down nos LEDs
P1OUT &= ~(BIT0 | BIT6);// Define os resistores
como pull-down
// Configuração do sistema de clock (ACLK =
32.768 Hz)
BCSCTL3 |= LFXT1S_0;//Configura LFXT1 como
cristal de 32.768 Hz
// Aguarda estabilização do oscilador
while (BCSCTL3 & LFXT1S_2);//espera oscilador
ficar pronto
// Configuração do Watchdog Timer para gerar um
intervalo de 1
WDTCTL = WDTPW | WDTTMSEL | WDTSSEL | WDT_
ADLY_1000;//ACLK para 1s
IE1 |= WDTIE;//Habilita interrupção do WDT
```

```
_bis_SR_register(GIE); // Habilita interrupções
globais
      while (1) { ___low_power_mode_3(); }//
      Reduz consumo e aguarda interrupção
}
#pragma vector=WDT_VECTOR//Rotina de interrupção
do Watchdog Timer
___interrupt void WDT_ISR (void) {
P1OUT ^= BIT0 | BIT6; //Alterna os LEDs a cada
1 segundo
}
```

REFERÊNCIAS BIBLIOGRÁFICAS

PEREIRA, Fábio. *Microcontroladores:* família MSP430 – Teoria e prática. 1. ed. São Paulo: Érica, 2005.

REIS, Francisco Everton Uchôa. *O Microcontrolador MSP430G2553*: Recursos básicos Memória RAM e Flash, sistema de clock e reset, pinos e portas com exemplos de utilização. Módulo I. Teresina: EDUFPI, 2025.

TEXAS INSTRUMENTS. *MSP430G2553 Mixed Signal Microcontroller (Rev. H)*. Dallas, TX: Texas Instruments, 2013.

TEXAS INSTRUMENTS. *MSP430F2xx, MSP430G2xx Family user's guide*. Dallas, dez. 2004. Disponível em: https://www.ti.com/lit/ug/slau144j/slau144j.pdf. Acesso em: 12 jun. 2025.

TEXAS INSTRUMENTS. *MSP430F2xx, MSP430G2xx Family user's guide* (Rev. K). Dallas, [s.d.]. Disponível em: https://www. ti.com/lit/ug/slau144k/slau144k.pdf. Acesso em: 12 jun. 2025.