4 Interrupções com o MSP430

Versão 1.0

A interrupção é um recurso que traz muita flexibilidade aos sistemas processados. Graças a ela eventos importantes podem receber atenção imediata do processador. A interrupção, como o próprio nome diz, interrompe ao fluxo normal de processamento da CPU e desvia a execução para um endereço previamente especificado. Podemos comparar a interrupção a uma chamada de sub-rotina, só que ao invés de usar uma instrução do tipo `CALL`, a chamada é feita por um evento gerado por *hardware*. Dizemos assim que interrupção é uma chamada de sub-rotina ativada por *hardware*. À sub-rotina que atende a uma interrupção damos o nome de Rotina Serviço de Interrupção, ISR do inglês *Interrupt Service Routine* ou também *Interrupt Handler*. Como toda sub-rotina, ela deve terminar com uma instrução de retorno, que no caso do MSP é a instrução *assembly* `RETI`.

4.0. Quero Usar Interrupções do MSP430 e Não Pretendo Ler Todo Este Capítulo

A interrupção pode ser entendida como uma "chamada de sub-rotina" por *hardware*. Não existe o `CALL`, apenas o evento que leva o MSP a executar uma determinada sub-rotina. O fabricante previu uma série de eventos de *hardware* que podem invocar sub-rotinas, tais como alteração no nível lógico de uma porta, *overflow* num *timer*, término do envio de um dado por uma porta serial e muitos outros. Para cada evento que pretende usar, o

programador precisa indicar o endereço da sub-rotina a ser chamada. Isso é feito preenchendo a Tabela de Vetores de Interrupção (Tabela 4.1) com os endereços das suas sub-rotinas.

Essa tabela fica armazenada na memória *Flash* e por isso, precisa ser especificada previamente, junto com o programa. O *loader* tem o dever de carregar o programa do usuário e a Tabela de Vetores de Interrupção. Há uma forma específica de indicar o endereço da sub-rotina a ser carregada numa determinada posição da tabela. O exemplo abaixo indica que a posição 42 da Tabela de Vetores de Interrupção deve receber o endereço da sub-rotina `P2_ISR`. Para mais exemplos, veja os exercícios resolvidos.

```
            .sect ".int42"          ;Vetor da porta P2
            .short  P2_ISR          ;Endereço a ser armazenado
```

A sub-rotina que atende a uma interrupção precisa terminar com a instrução RETI e tem a obrigação de manter intacto o contexto do programa principal. Se a sub-rotina de interrupção precisa usar um registrador que também está sendo usado pelo programa principal, ela tem a obrigação de guardar o valor desse registrador (em geral para a pilha) antes de usá-lo e, depois, a obrigação de restaurar seu valor antes de retornar.

No MSP toda interrupção é gerada a partir de um determinado *bit* de um certo registrador de controle. O nome dado a esses *bits* sempre termina com as letras IFG (*Interrupt Flag*). Para cada interrupção, existe uma forma do programador controlar sua habilitação, o que é feito com um *bit* adicional, cujo nome termina com as letras IE (*Interrupt Enable*).

Ao examinar a Tabela de Vetores de Interrupção (Tabela 4.1) o leitor vai constatar que existem posições que são exclusivas para um determinado evento e que existem posições que são compartilhadas com diversos eventos. No caso de uma interrupção exclusiva, o pedido de interrupção é apagado automaticamente que se inicia a execução de sua sub-rotina. Já para os vetores compartilhados, o programador precisa consultar um registrador auxiliar denominado de Registrador Vetor de Interrupção (nome sempre termina em IV, de *Interrupt Vector*) para saber quem, dentre os possíveis candidatos, provocou a interrupções. Veja a listagem disponível no tópico 4.6.

Para finalizar, é preciso informar que o Registrador de *Status* (SR) tem o bit GIE (*General Interrupt Enable*) que é a habilitação geral das interrupções. Se GIE = 0, nenhuma interrupção ocorre. Se GIE = 1, as interrupções que foram habilitadas podem acontecer.

O que se fez neste tópico foi uma abordagem muito pobre sobre as interrupções. É fortemente recomendado que o leitor estude o resto deste capítulo.

4.1. Introdução

Para melhor explicar o que é interrupção e sua versatilidade vamos considerar uma situação hipotética em que o MSP precisa receber vários dados (*bytes*). Consideremos o caso da Figura 4.1, no qual os dados chegam por um barramento denominado “Dados” e o instante de validade de cada dado é indicado pelo flanco de descida de uma linha denominada #STRB. Note que o sinal #STRB não é periódico.

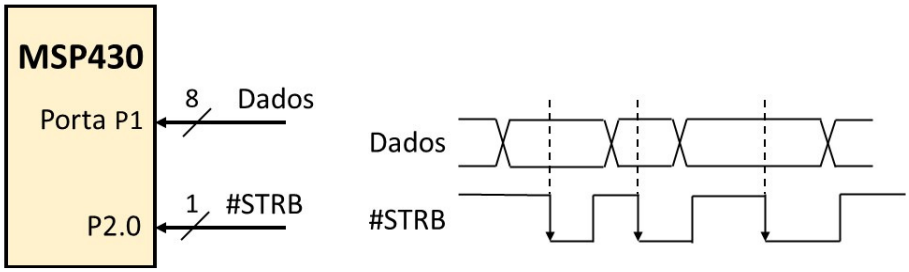

Figura 4.1. Esquema para o MSP receber uma sequência de dados e o instante de validade de cada dado, caracterizado pelo flanco de descida do sinal #STRB.

A proposta de um algoritmo para a recepção dos dados segundo este protocolo é bem simples.

1. Enquanto STRB = 1, ficar esperando;
2. Ler o dado;
3. Enquanto STRB = 0, ficar esperando;
4. Voltar ao passo 1;

A listagem abaixo apresenta a sugestão de uma sub-rotina em *assembly* para receber esses dados. Observe os dois laços marcados em cinza. Eles servem para que se aguarde uma transição na linha #STRB. O primeiro laço (LB1) aguarda pelo flanco de descida e o segundo laço (LB2) aguarda pelo flanco de subida. Essa técnica é denominada *polling* (consulta, em português). Ela é simples e rápida, porém note que não sobra tempo para mais nada. A CPU é completamente consumida no monitoramento da linha #STRB.

Proposta de uma sub-rotina para recepção dos dados segundo a Figura 4.1

```
; Subrotina -- REC -- para receber dados
; Inicia com: R5 = Quantidade de dados
;             R6 = Endereço inicial para armazenar os dados

REC:  MOV.B       #0,&P1DIR         ;Porta P1 como entrada
      MOV.B       #0xFF,&P1REN      ;Habilitar resistores de P1
      MOV.B       #0xFF,&P1OUT      ;Selecionar Pullup
```

```
      BIC.B     #BIT0,&P2DIR      ;Pino P2.0 como entrada
      BIS.B     #BIT0,&P2REN      ;Habilitar resistor
      BIS.B     #BIT0,&P2OUT      ;Selecionar pullup

LB1:  BIT.B     #BIT0,P2IN        ;Testar P2.0
      JNZ       LB1               ;Se P2.0=1, repetir laço
      MOV.B     &P1IN,0(R6)       ;Armazenar dado
      INC.W     R6                ;Incrementar ponteiro de dados
      DEC.W     R5                ;Decrementar contador
      JZ        FIM               ;Se contador=0, fim
LB2:  BIT.B     #BIT0,P2IN        ;Testar P2.0
      JZ        LB2               ;Se P2.0=0, repetir laço
      JMP       LB1               ;P2.0=1, repetir tudo
FIM   RET
```

Vamos agora considerar uma situação hipotética em que o MSP precisa receber *bytes* de acordo com o protocolo da Figura 4.1 e, ao mesmo tempo, realizar algumas outras tarefas como, por exemplo, executar duas sub-rotinas denominadas `ROT1` e `ROT2`. Uma solução seria chamar essas sub-rotinas enquanto a CPU está esperando por uma alteração na linha #STRB, como apresentado abaixo. A cada consulta de P2.0, rodam-se as duas sub-rotinas. Se a execução de `ROT1` e `ROT2` for rápida e o pulso #STRB for lento, provavelmente vai funcionar. Entretanto, se uma das sub-rotinas demorar, perderemos um dado ou faremos a leitura no instante errado.

Sugestão para alteração do laço LB1

```
LB1:  CALL      #ROT1             ;Chamar ROT1
      CALL      #ROT2             ;Chamar ROT2
      BIT       #BIT0,P2IN        ;Testar P2.0
```

Esse é o problema básico com o *polling*. Para termos certeza de que todos os dados serão recebidos, a CPU precisa ficar exclusivamente dedicada ao *polling*. Só podemos permitir a saída temporária do *polling* se tivermos a certeza da temporização dos sinais (Dados e #STRB) e do tempo máximo consumido por cada sub-rotina. Vem agora uma pergunta: será que não podemos condicionar a execução de uma sub-rotina à alteração do nível lógico de um pino do MPS?

A resposta a essa pergunta é sim e o nome usado é interrupção. Vamos continuar com o exemplo, mas agora condicionar a execução de um trecho de programa à presença de um flanco de descida em P2.0, como mostrado na próxima listagem (algumas simplificações foram feitas). Nesta listagem, as três linhas marcadas em cinza indicam (ainda de forma misteriosa) ao processador que a sub-rotina `ISR` deve ser executada a cada flanco de descida de P2.0.

Agora, a solução ficou simples. Deixamos o processador num laço executando as sub-rotinas `ROT1` e `ROT2`. Cada vez que surgir um flanco de descida em P2.0, o laço principal

`LB1` é interrompido, a sub-rotina `P2_INT` é executada e faz a recepção do *byte*. Ao terminar a sub-rotina de interrupção, a execução segue a partir do ponto onde foi interrompida. É importante comentar que cada vez que a sub-rotina `P2_INT` é executada, há a certeza de que acabou de acontecer um flanco de descida em P2.0. O leitor deve ter notado o uso da instrução `RETI` ao final da rotina `P2_INT`. Este é um retorno especial que indica o fim de uma rotina de interrupção. Note que o laço principal também monitora R5 e quando ele chega a zero é porque terminou a recepção programada.

O conceito básico que se quer passar é o de que o programa (no caso, laço LB1) fica dedicado às suas tarefas e de vez em quando, por ação do *hardware*, é interrompido para execução de alguma rotina e, quando esta termina, volta à execução normal, a partir do ponto onde foi interrompido.

Proposta de uma sub-rotina para recepção dos dados por interrupção

```
; Subrotina -- REC -- para receber dados
; Inicia com: R5 = Quantidade de dados
;             R6 = Endereço inicial para armazenar os dados

REC:  MOV.B       #0,&P1DIR         ;Porta P1 como entrada
      MOV.B       #0xFF,&P1REN      ;Habilitar resistores de P1
      MOV.B       #0xFF,&P1OUT      ;Selecionar Pullup

      BIC.B       #BIT0,&P2DIR      ;Pino P2.0 como entrada
      BIS.B       #BIT0,&P2REN      ;Habilitar resistor
      BIS.B       #BIT0,&P2OUT      ;Selecionar pullup
      BIS.B       #BIT0,&P2IE       ;Habilita interrupção em P2.0
      BIS.B       #BIT0,&P2IES      ;Seleciona flanco de descida em P2.0
      EINT                          ;(GIE=1) Habilita as interrupções

LB1:  CALL        #ROT1             ;Chamar ROT1
      CALL        #ROT2             ;Chamar ROT2
      TEST        R5                ;R5=0?
      JNZ         LB1               ;Repetir ser R6 diferente de zero
      RET

; Sub-rotina para atender às interrupções
P2_INT:
      MOV.B       &P1IN,0(R6)       ;Armazenar dado
      INC.W       R6                ;Incrementar ponteiro de dados
      DEC.W       R5                ;Decrementar contador
      RETI                          ;(RETI) Retorno de interrupção
```

4.2. Conceitos Básicos sobre Interrupções

Antes de prosseguirmos com o estudo das interrupções do MSP, vamos explicar alguns conceitos importantes sobre:

- A geração do pedido de interrupção;
- A habilitação e a desabilitação da interrupção;
- A sub-rotina a ser chamada por uma interrupção;
- A interrupção e o programa interrompido;
- O endereço de regresso de uma interrupção;
- A guarda do contexto do programa interrompido;
- A precedência entre interrupções e
- O Registrador de *Status*.

O primeiro conceito trata do evento gerador do pedido de interrupção. Diversos eventos podem gerar um pedido de interrupção, por exemplo, um nível baixo num determinado pino do MSP, ou o *overflow* de um Temporizador/Contador, a recepção ou a transmissão de um *byte* pela porta serial e muitos outros. Praticamente, todas as interrupções do MSP têm o início marcado por um determinado *bit* indo para 1, esse *bit* costuma ser chamado de *flag* de interrupção. Existem diversas *flags* de interrupção, uma para cada recurso que pode interromper a CPU. O leitor pode identificar essas *flags* nos diversos registradores do MSP, porque elas sempre terminam com as letras IFG. Vamos sempre indicá-las ao longo deste texto. Por exemplo, a *flag* que provoca a interrupção do Timer A é denominada TA**IFG**.

Para cada interrupção é preciso prever uma forma de habilitá-la e desabilitá-la. Quando a interrupção está desabilitada, mesmo que ocorram pedidos, eles são ignorados. Normalmente, a habilitação é controlada através dos *bits* de algum registrador. Por exemplo, no caso do programa acima, a interrupção somente pode acontecer se o *bit* P2IE.0 estiver em 1. É costume usar a palavra máscara para se referir à desabilitação. Assim, usa-se o termo mascarar uma interrupção para indicar que se está desabilitando a interrupção. A desabilitação é importante porque, muitas vezes, não se deseja atender uma determinada interrupção ou porque existem trechos de programa que são críticos e que não devem ser interrompidos.

É preciso, de alguma forma, especificar o endereço da sub-rotina a ser chamada para cada interrupção. Esse endereço recebe o nome de Vetor de Interrupção, já que ele aponta para o local onde está a sub-rotina a ser chamada. Essa sub-rotina que atende a uma interrupção é denominada de Rotina Serviço de Interrupção, abreviada pela sigla ISR (do inglês *Interrupt Service Routine*), ou simplesmente, Rotina de Interrupção. Algumas vezes usa-se o nome *Interrupt Handler*. Assim, para cada interrupção é preciso especificar o endereço de sua rotina de serviço. Algumas arquiteturas usam vetorização fixa, ou seja, para cada interrupção existe um endereço previamente definido. Nesse caso, a definição é feita pelo fabricante e nunca pode ser mudada. Outras arquiteturas trabalham com vetores flutuante (móveis), ou seja, através de uma tabela em memória o usuário define o endereço da sub-rotina de serviço para cada interrupção. Por exemplo, o computador PC, assim como o MSP trabalham com uma tabela de vetores de interrupção flutuante. Vide Tabela 4.1.

Para o programa que está em execução, denominado Programa Principal, a interrupção é um evento assíncrono, ou seja, pode interrompê-lo a qualquer momento. Entretanto, é preciso ficar bem claro que a interrupção só é aceita ao final da execução de uma instrução de máquina, ou seja, a interrupção não interrompe a execução de uma instrução (*assembly*). A lógica de controle da CPU espera terminar a instrução que está sendo executada para aceitar a interrupção e desviar para a sub-rotina de serviço. Antes do desvio, ela armazena na pilha o endereço de regresso. Ao terminar a sub-rotina de interrupção (instrução `RETI`), esse endereço é retirado da pilha e a execução retoma o fluxo original.

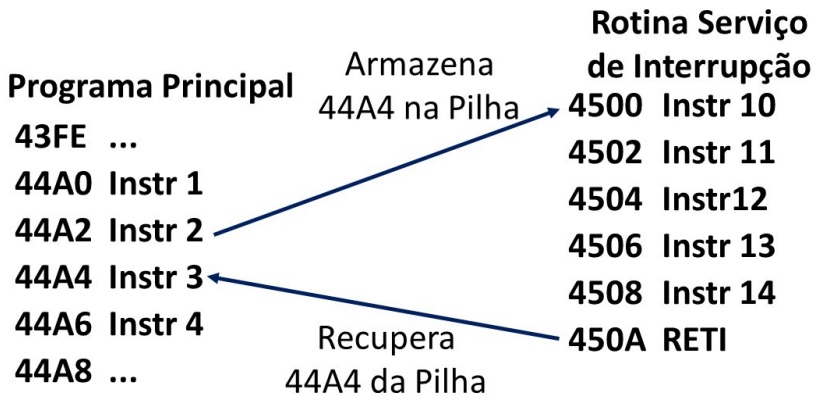

Figura 4.2. Ilustração de um pedido de interrupção que chegou enquanto estava sendo executada a Instrução 2 (endereço 0x44A2).

A Figura 4.2 ilustra o caso de um pedido de interrupção chegando no exato momento em que estava sendo executada a Instrução 2, que está no endereço 0x44A2. O processador termina a execução dessa instrução, armazena na pilha o endereço de regresso (0x44A4), que é o endereço da próxima instrução e copia da Tabela de Vetores de Interrupção para o Contador de Programa o endereço (previamente definido) da sub-rotina de serviço da interrupção que, para esse exemplo, é suposto ser igual a 0x4500. Ao terminar a rotina de interrupção, a execução da instrução de retorno de interrupção (`RETI`) faz o processamento retomar a partir do endereço que estava armazenado no topo da pilha, que é o endereço 0x44A4, neste exemplo. Assim, após terminar a rotina serviço de interrupção o processamento continua exatamente a partir do ponto onde havia sido interrompido.

Note que, em geral, o programa principal não “percebe” que houve interrupção. Por isso, pode haver confusão se a sub-rotina de interrupção usar os mesmos registradores (recursos) que o programa principal. Para o programa principal, esses registradores mudariam de valor como que por mágica. Por isso, a sub-rotina de interrupção tem a

obrigação de armazenar todos os registradores que irá utilizar (e que também estão sendo usados pelo programa principal), para que antes de retornar, restaure o contexto do programa principal. É usual que no início da sub-rotina que atende a uma interrupção se coloquem instruções para salvar os registradores importantes e ao final, são inseridas instruções para restaurar os registradores e a finalização é feita com a instrução `RETI`. A Figura 4.3 ilustra o caso em que os registradores R5 e R6 são usados pelo programa principal e pela rotina de interrupção. Neste caso, a rotina de interrupção guardou na pilha cópias de R5 e R6. Assim, ela pode usar como quiser esses dois registradores. Antes de retornar da execução, ela restaura os valores de R5 e R6. Note que o armazenamento e a recuperação de valores na pilha são feitos em sequência invertida (`PUSH R5`, `PUSH R6`, ..., `POP R6`, `POP R5`).

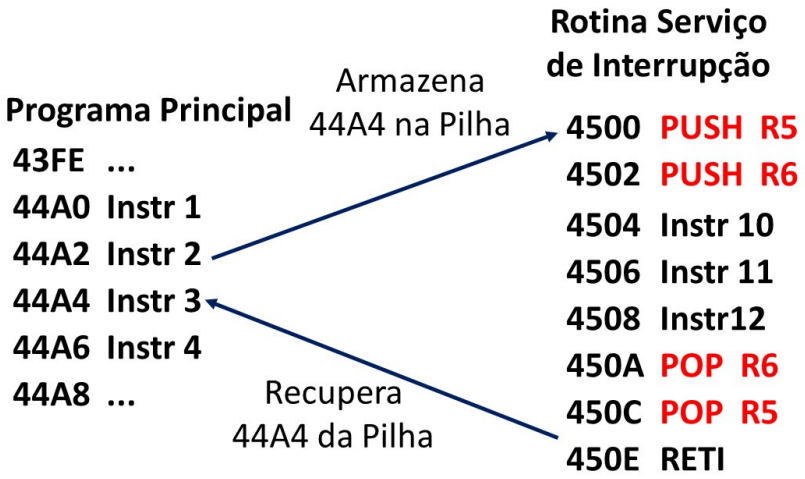

Figura 4.3. Exemplo de Rotina Serviço de Interrupção que guarda na pilha cópias de R5 e R6, que são restauradas antes do regresso.

Outro ponto importante é explicar o que acontece se chegar um novo pedido de interrupção enquanto se está atendendo o atual? Essa pergunta é respondida com a seguinte frase: no MSP, uma interrupção não interrompe outra interrupção. Isto significa que esse pedido fica pendente e só será aceito após a execução da instrução `RETI`. Esta instrução `RETI` é importante porque ela indica ao processador que terminou interrupção e que já pode aceitar a próxima. Se o programador se equivocar e usar a instrução `RET` no lugar de `RETI`, o processador vai se perder. No caso de um `RET`, a cópia do registrador de *Status* (SR) que estava na pilha será copiada para o Contador de Programa e com isso a execução fica imprevisível. Veja detalhes na Figura 4.6.

Para resolver o caso de dois pedidos que chegam ao mesmo tempo, existe uma precedência interna: as interrupções de maior número são aceitas primeiro. A numeração está apresentada na Tabela 4.1.

É importante examinar novamente o Registrador de *Status*. Um estudo detalhado já foi apresentado no item 1.3 do Capítulo 1, especificamente na Figura 1.5. Entretanto, para fins de clareza, repetimos abaixo a figura desse registrador onde está destacado o *bit* GIE, que é a Habilitação Geral das Interrupções.

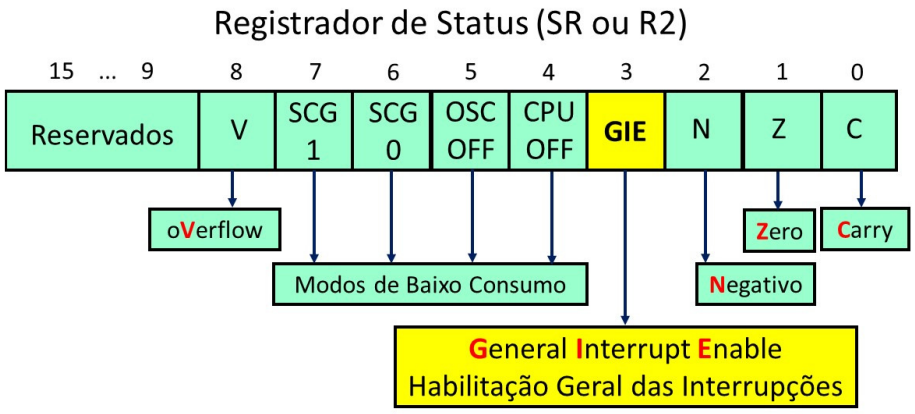

Figura 4.4. Composição do Registrador de Status do MSP430.

O *bit* 3 desse registrador, denominado "GIE", é a Habilitação Geral das Interrupções. Com GIE = 0, nenhuma interrupção é aceita. Por outro lado, com GIE = 1, uma interrupção pode ser aceita, desde que sua habilitação individual esteja ativada. Ao entrar numa sub-rotina de interrupção, o *hardware* zera este o *bit* GIE, para evitar que outras interrupções ocorram. A final da sub-rotina de interrupção, a instrução de retorno (`RETI`) ativa este *bit* para permitir que as interrupções subsequentes possam ocorrer. Mais adiante, estudaremos isto com mais detalhes. As instruções assembly `EINT` (ativar) e `DINT` (zerar) permitem que programador controle este *bit*.

4.3. Vetores de Interrupção do MSP430F5529

Apresentamos a seguir a Tabela de Vetores de Interrupção para as 23 interrupções do MSP430F5529. A quantidade de vetores usados varia de acordo com a versão do microcontrolador. Como já afirmamos, nesta tabela se armazena o endereço da sub-rotina (ISR) a ser chamada para cada um dos eventos possíveis. (pag. 20, Manual do MSP430F5529). De acordo com a interrupção que pretende usar, o programador deve colocar nesta tabela, numa determinada posição, o endereço da sub-rotina a ser chamada. No caso do exemplo de recepção de dados pela porta P1, usando a interrupção por P2.0, é preciso gravar nas posições 0xFFD4 e 0xFFD5 o endereço de início da sub-

rotina `P2_INT`. O montador do CCS oferece recursos para auxiliar o programador neste sentido. Veremos isso mais adiante.

Tabela 4.1. Tabela de Vetores de Interrupção do MSP430F5529

Prioridade	**Fonte**	**Endereço (16 *bits*)**	
		MSB	**LSB**
63 (mais alta)	*Reset*	0xFFFF	0xFFFE
62	NMI Sistema	0xFFFD	0xFFFC
61	NMI Usuário	0xFFFB	0xFFFA
60	Comparador B	0xFFF7	0xFFF8
59	Timer TB0 (CCIFG0)	0xFFF6	0xFFF6
58	Timer TB0 (demais)	0xFFF4	0xFFF4
57	Watchdog Timer	0xFFF2	0xFFF2
56	USCI A0	0xFFF1	0xFFF0
55	USCI B0	0xFFEF	0xFFEE
54	Conversor AD	0xFFED	0xFFEC
53	Timer TA0 (CCIFG0)	0xFFEB	0xFFEA
52	Timer TA0 (demais)	0xFFE9	0xFFE8
51	Porta USB	0xFFE7	0xFFE6
50	DMA	0xFFE5	0xFFE4
49	Timer TA1 (CCIFG0)	0xFFE3	0xFFE2
48	Timer TA1 (demais)	0xFFE1	0xFFE0
47	Porta P1	0xFFDF	0xFFDE
46	USCI A1	0xFFDD	0xFFDC
45	USCI B1	0xFFDB	0xFFDA
44	Timer TA2 (CCIFG0)	0xFFD9	0xFFD8
43	Timer TA2 (demais)	0xFFD7	0xFFD6
42	Porta P2	0xFFD5	0xFFD4
41	Real-Time Clock	0xFFD3	0xFFD2
40	Reservada	0xFFD1	0xFFD0
...	Reservada	...	...
0 (mais baixa)	Reservada	0xFF81	0xFF80

Nesta Tabela 4.1 há uma grande quantidade de recursos que ainda não foram estudados e que serão abordados oportunamente. No final deste capítulo está uma versão mais detalhada desta tabela.

A tabela precisa de 128 *bytes*, pois são 64 posições, cada uma com 2 *bytes*. Ela é construída na memória *Flash* a partir do último endereço dentro da primeira página de 64 KB, ou seja, 0xFFFF. Como são 128 *bytes*, a tabela excursiona desde 0xFF80 até 0xFFFF. Já vimos que o MSP segue a arquitetura *Big-Endian*, isto significa que, de uma palavra de 16 *bits*, armazena-se primeiro o LSB e depois o MSB (segundo o sentido crescente dos endereços).

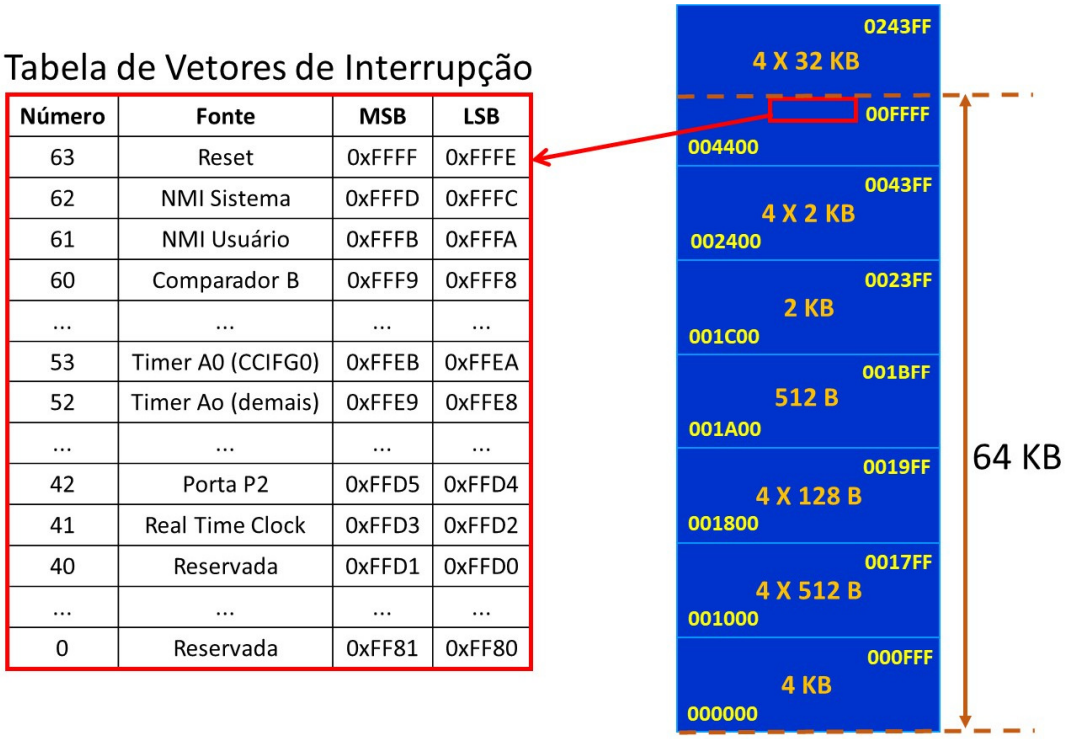

Figura 4.5. Localização na memória Flash da Tabela de Vetores de Interrupção.

Ainda sobre essa Tabela de Vetores de Interrupção, é oportuno indicar duas interrupções que não podem ser desabilitadas e que por isso são denominadas *Non-Maskable Interrupt*, donde surge a sigla NMI. Elas são: a do sistema e a do usuário. Tais interrupções são dedicadas a eventos importantes, relacionados principalmente com a alimentação da CPU, falha no oscilador, violação de acesso à memória etc. Por hora seu estudo não se faz necessário.

Observação:
A Tabela de Vetores de Interrupção está construída na memória *Flash* do MSP430 e por isso precisa ser carregada junto com o programa do usuário. O programador não tem como alterar esta tabela durante a execução de suas rotinas. Entretanto, o *bit* SYSRIVECT do registrador SYSCTL, se colocado em nível alto, força a construção desta tabela a partir do topo da SRAM. Neste caso, os programas podem alterá-la. O *reset* sempre configura para a tabela seja construída na memória *Flash*.

4.4. Passos na Aceitação e Retorno de uma Interrupção

Agora que já temos uma boa ideia do que é uma interrupção, podemos estudar a sequência executada pelo MSP430, quando uma interrupção é aceita. Vamos considerar que o *bit* GIE esteja em 1, que a interrupção está habilitada e que seu pedido é efetivado através do respectivo *flag* de interrupção (xxxIFG). Quando isto acontece, é realizada a sequência de ações listada abaixo.

1. A instrução que estiver em execução é finalizada.
2. O contador de programa (PC), que sempre aponta para a próxima instrução, é guardado na pilha.
3. O Registrador de *Status* (SR) é guardado na pilha.
4. A interrupção de maior prioridade (caso haja múltiplos pedidos pendentes) é selecionada.
5. Se a *flag* que provocou a interrupção for única, ela é apagada.
6. O Registrador de *Status* (SR) é zerado. Isto implica que o *bit* GIE também é zerado, o que impede que novas interrupções sejam aceitas.
7. Uma das posições da Tabela de Vetores de Interrupção (de acordo com o pedido de interrupção) é copiada para o Contador de Programa (PC). A execução continua a partir deste endereço.

A execução da sequência apresentada gasta 6 períodos de relógio e recebe o nome de Latência de Interrupção. Em outras palavras, o intervalo de tempo entre o momento em que a CPU aceita a interrupção e o momento em que a primeira instrução da rotina de interrupção começa a ser executada é de 6 períodos do relógio da CPU (MCLK).

É interessante ver o uso da pilha pelo processo de interrupção. A Figura 4.6 apresenta o estado da pilha antes e depois, quando a rotina de interrupção começa a ser executada. Como o Contador de Programa (PC) é armazenado na pilha, ao retornar da interrupção, a execução continua a partir da próxima instrução. Veja que o Registrador de *Status* também é guardado na pilha, com isso, as *flags* de *Carry*, Zero, etc que, porventura, estavam em uso pelo programa interrompido, são preservadas. Veremos que o regresso de uma interrupção restaura este registrador.

Um ponto muito importante é que após guardar na pilha uma cópia do Registrador de *Status* (SR), ele é zerado. Com isso, a *flag* de Habilitação Geral das Interrupções (GIE) vai para zero, o que impede que novas interrupções sejam aceitas. Por isso, uma interrupção não interrompe outra interrupção. Quando se retorna da interrupção, o registrador SR é restaurado com seu valor original e novas interrupções podem ser aceitas.

É importante comentar que ao zerar o registrador SR, a CPU sai do modo de economia de energia, caso um desses modos estivesse em uso. Isto será estudado mais adiante. É claro que o programador pode, dentro da rotina que atende a interrupção, ativar a *flag* GIE. Com isso, a partir deste ponto, mesmo "dentro da rotina de interrupção", uma nova interrupção pode ser aceita. Este recurso se chama Aninhamento de Interrupções (*Interrupt Nesting*) e só deve ser usado quando realmente for necessário.

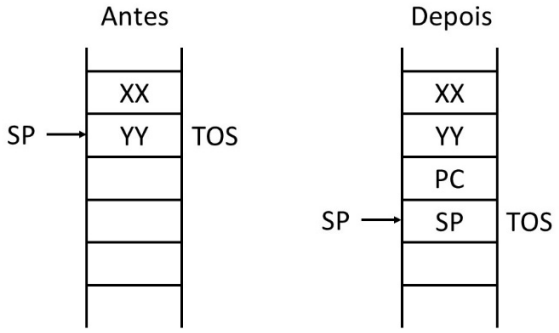

Figura 4.6. Situação da pilha antes da interrupção e após o desvio para a interrupção.

O retorno da rotina que atende a uma interrupção é feito com a instrução RETI, que dispara a sequência abaixo.

1) O Registrador de *Status* (SR) é atualizado (POP SR) com a cópia que ficou guardada ne pilha.
2) O Contador de Programa (PC) é atualizado (POP PC) com a cópia que ficou na pilha e com isso a execução segue a partir da instrução seguinte a que foi interrompida.

A latência do retorna da interrupção é de 5 períodos de relógio da CPU (MCLK). Isto significa que 5 ciclos após a instrução `RETI`, a CPU inicia a execução da próxima instrução do programa principal. É importante comentar que ao restaurar SR com sua cópia que estava na pilha, a CPU volta para o mesmo estado de consumo de energia que estava em uso antes da interrupção. Isto será estudado em um outro capítulo.

Alguns processadores forçam a execução de uma instrução do programa principal após o retorno de uma interrupção, antes de aceitar uma nova interrupção. Este não é o caso do

MSP430. Assim, se por algum equívoco uma *flag* de interrupção não for apagada dentro da rotina de interrupção, a CPU ficará eternamente presa executando esta rotina.

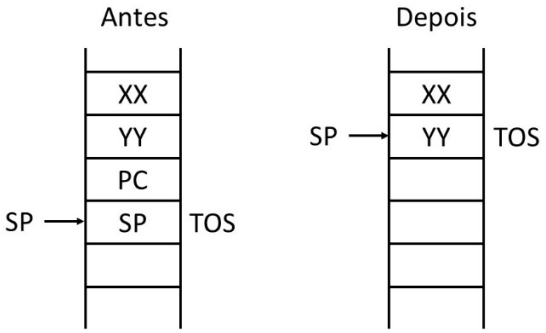

Figura 4.6. Situação da pilha antes e após o retorno da interrupção.

No Capítulo 1, vimos que o MSP430 tem sua arquitetura baseada em um *pipeline* de 3 estágios. Isto significa que quando a execução de uma instrução é iniciada, uma segunda está sendo decodificada e uma terceira está sendo buscada na memória. Assim, a instrução que está em execução pode não ter efeito sobre aquela que está sendo decodificada. Por isso, o fabricante recomenda alguns cuidados na habilitação (`EINT`) e desabilitação (`DINT`) das interrupções.

- A instrução seguinte à instrução que habilita GIE (`EINT`), sempre é executada, mesmo que haja interrupções pendentes.
- Inclua, pelo menos, uma instrução entre a que apaga a habilitação de uma interrupção ou a *flag* de interrupção e a instrução `EINT`. Sugestão: coloque um `NOP` antes da instrução `EINT`.
- Inclua pelo menos uma instrução entre a instrução `DINT` e o trecho de código que necessita de proteção contra interrupções. Sugestão: coloque um `NOP` após a instrução `DINT`.
- Nunca ative e apague o *bit* GIE em sequência, ou seja, nunca a sequência `EINT` – `DINT`. Use um `NOP` entre elas.

Vamos ver que o montador é ainda mais cuidadoso e sugere que as instruções `EINT` e `DINT` sejam envolvidas por 2 `NOPs`, um ante e outro depois.

4.5. Interrupções com o MSP430

Agora que já temos uma boa ideia sobre interrupções, vamos apresentar mais algumas particularidades de seu funcionamento. A Figura 4.7 apresenta um diagrama ilustrativo das interrupções com o MSP430. Vimos que a habilitação geral, feita pelo *bit* GIE do

Registrador de *Status*. Nenhuma interrupção ocorre se GIE = 0. Para cada interrupção, existe uma habilitação particular feita por um *bit*, que na figura estão denominados de A**IE**, B**IE** etc. Na denominação desses *bits*, o fabricante sempre usa as letras IE como as duas últimas letras (IE do inglês *Interrupt Enable*). Para que a interrupção ocorra é necessário um evento. Por evento, como já vimos, entendemos a chegada de um *byte* pela porta serial, a mudança de estado num determinado pino, o *overflow* num *timer* etc. Na figura usamos os nomes: Evento A, Evento B, ..., Evento Z. Quando um desses eventos ocorre, ele ativa uma *flag*. Na figura marcamos essas *flags* com AIFG, BIFG, ..., ZIFG. O fabricante sempre termina a denominação de tais *flags* com as letras IFG (do inglês *Interrupt Flag*).

Agora, vamos a alguns detalhes, tomando como exemplo toda a sequência que acontece com o Evento A. Quando o Evento A ocorre, ele ativa a *flag* AIFG. Se a habilitação deste evento (AIE = 1) estiver ativa e se a habilitação geral estiver ativa (GIE = 1), então acontece a interrupção que na figura denominamos INTA. O desvio para a sub-rotina de interrupção segue toda a sequência que já estudada.

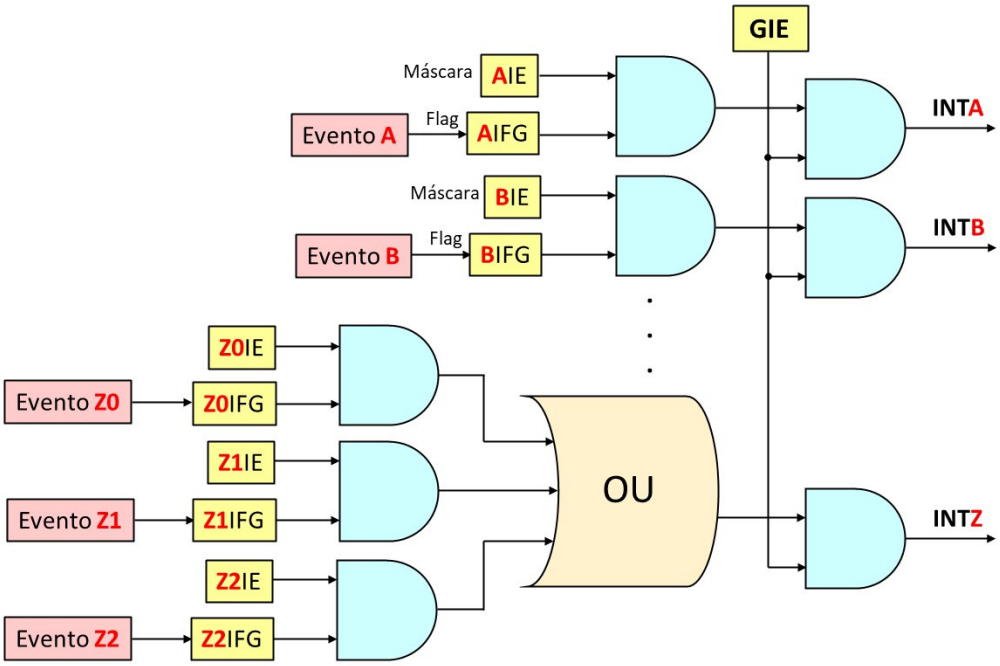

Figura 4.7. Diagrama ilustrativo das interrupções no MSP430.

Vamos agora estudar o caso em que vários eventos compartilham uma única interrupção, como é o caso da interrupção INTZ da figura. Os eventos que podem provocar a interrupção Z foram denominados de Evento Z0, Evento Z1 e Evento Z2. Temos então 3 eventos que podem provocar a mesma interrupção. A rotina que atende a esta

interrupção precisa, portanto, consultar as *flags* Z0IFG, Z1IFG e Z2IFG para determinar qual deles provocou a interrupção.

Essa busca para determinar quem, dentre os candidatos, provocou a interrupção pode consumir muito tempo. Imagine o caso em que existam 8 candidatos. Para tornar essa busca rápida, o MSP430 faz uso dos chamados Registradores de Vetor de Interrupção que indica ao programador qual a origem da interrupção. Na denominação desses registradores o fabricante sempre usa IV (do inglês *Interrupt Vector*) como as duas últimas letras do nome. No caso do exemplo, vamos inventar o registrador ZIV. Como a finalidade deste registrador é indicar a origem da interrupção, ele tem alguns recursos extras. Para poder explicá-los, vamos imaginar que o Evento Z0 seja o mais prioritário e, Z2, o menos prioritário. Então, propomos um registrador especial cujo funcionamento está ilustrado na tabela abaixo.

Tabela 4.2. ZIV: Registrador Vetor de Interrupção do recurso Z

Prioridade	Valor lido	Significado
-	0	Nenhuma interrupção pendente
Maior	2	Z0IFG pediu interrupção
-	4	Z1IFG pediu interrupção
Menor	6	Z2IFG pediu interrupção

O hipotético registrador ZIV retorna um número par, indicando qual dos candidatos provocou a interrupção. No caso de duas interrupções pendentes, ele retorna o número correspondente à interrupção de maior prioridade. Um outro recurso interessante é que ao ser lido, o *bit* relatado é apagado. Por exemplo, digamos que temos dois pedidos simultâneos, indicados por Z0IFG = 1 e Z2IFG = 1. A primeira leitura de ZIV vai retornar 2 e a *flag* Z0IFG é apagada. A segunda leitura vai retornar 6 e a *flag* Z2IFG é apagada. Uma terceira leitura de ZIV vai retornar 0. O número retornado é par para facilitar o salto relativo usando o PC. Veremos isto no próximo tópico.

Vamos agora apresentar uma regra geral que permite saber quando uma *flag* de interrupção é automaticamente zerada ao iniciar a correspondente sub-rotina de interrupção.

- Se a *flag* tiver um vetor exclusivamente para ela, ela é apagada ao iniciar a sub-rotina de interrupção.
- Se *flag* compartilha um vetor com diversas outras *flags*, ela é apagada quando o acesso ao seu registrador vetor de interrupção retornar seu número.

É claro que o programador pode acessar os diversos registradores de controle e apagar ou ativar qualquer uma dessas *flags* de interrupção.

Agora, Tudo vai ficar mais claro, pois vamos usar as portas P1 e P2 como exemplos de interrupção.

4.6. Interrupções com as Portas P1 e P2

No capítulo 3 estudamos as portas de I/O do MSP430. Elas são denominadas GPIO. As portas P1 e P2 possuem um recurso extra: elas podem provocar interrupções. São 4 os registradores dedicados às interrupções. Eles estão repetidos logo a seguir, onde *n* = 1 ou 2. Em resumo:

- PnIE → Habilita ou desabilita a interrupção por um *bit* de I/O da porta Pn.
- PnIES → Seleciona o tipo de flanco (evento) que vai disparar a interrupção.
- PnIFG → São os *flags* de interrupção. Registram se há interrupção pendente.
- PnIV → Registrador Vetor de Interrupção, indica a interrupção pendente de maior prioridade.

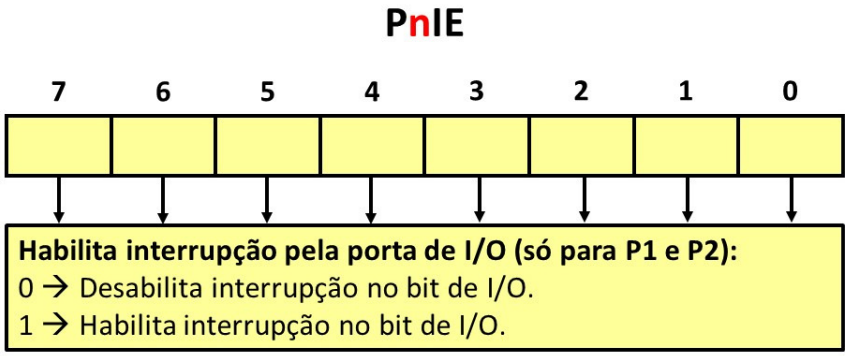

Figura 4.8. Registrador para habilitar ou desabilitar a interrupção por um bit de GPIO. Válido apenas para as portas P1 e P2, ou seja, n = 1 ou 2.

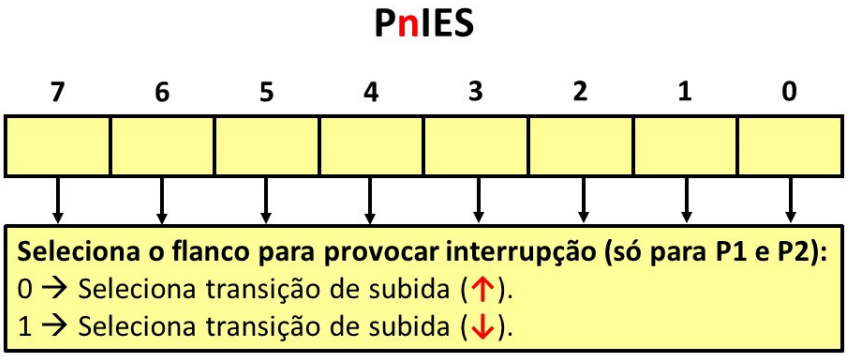

Figura 4.9. Registrador para indicar o tipo de flanco que provoca a interrupção. Válido apenas para as portas P1 e P2, ou seja, n = 1 ou 2.

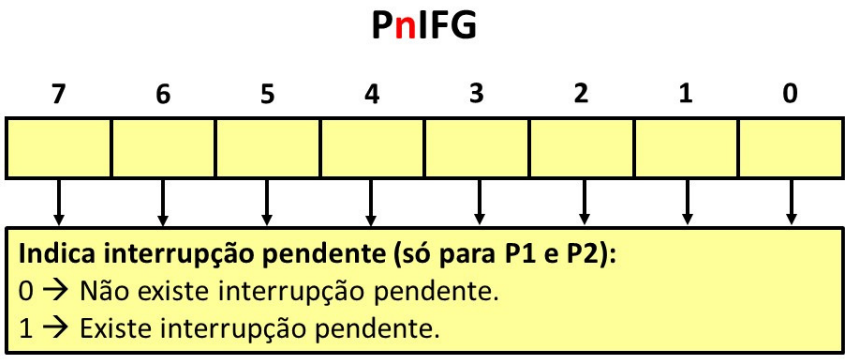

Figura 4.10. Registrador de flags de interrupção. Indica quais interrupções estão pendentes. Válido apenas para as portas P1 e P2, ou seja, n = 1 ou 2.

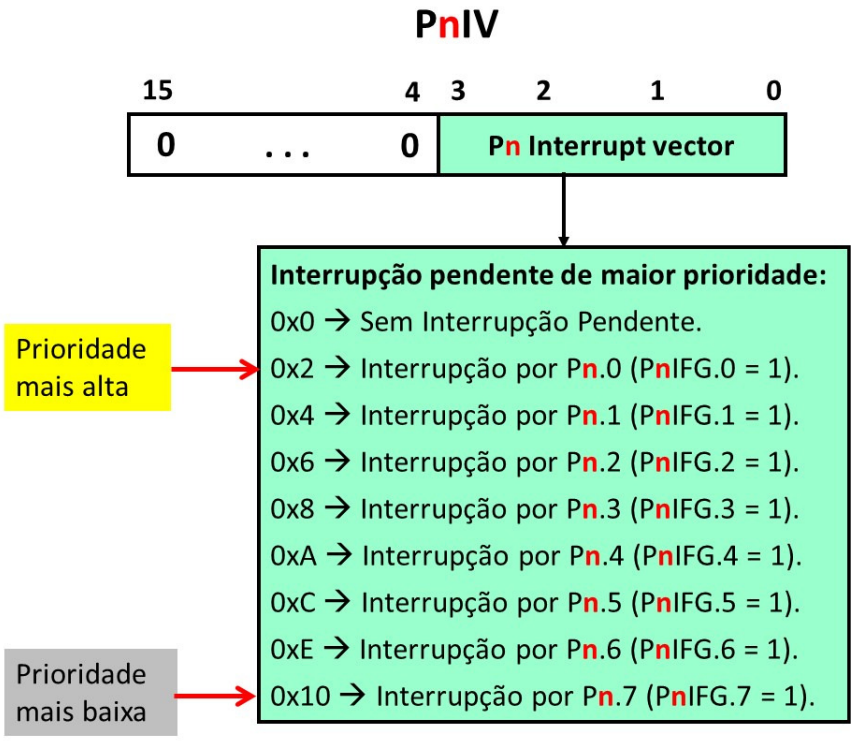

Figura 4.11. Registrador para indicar a interrupção pendente de maior prioridade. Válido apenas para as portas P1 e P2, ou seja, n = 1 ou 2.

Vamos agora escrever, de forma completa, a sub-rotina para receber dados, que foi usada como exemplo ilustrativo no início deste capítulo. Ela vai servir para expormos os conceitos finais de interrupção. Se o leitor está lembrado, é pedido que a cada flanco de descida em P2.0 o dado presente na porta P1 seja lido e armazenado na memória. Para escrever a solução, serão usados dois registradores:
R5 → contador de *bytes* recebidos e
R6 → ponteiro (endereço inicial) para guardar na memória os dados recebidos.

A listagem abaixo apresenta, de forma bem detalhada, a solução do problema proposto. É importante olhar com atenção alguns trechos desta sub-rotina. Marcado em cinza está o trecho que configura o *bit* P2.0 para provocar interrupção por flanco descendente. Note que antes de habilitar a interrupção, tivemos o cuidado de apagar o *bit* P2IFG.0. Isto apaga qualquer pedido anterior que porventura estivesse pendente. Esse cuidado se justifica porque a configuração da porta pode gerar flancos de subida ou descida e, inadvertidamente, invocar uma interrupção. Outra possibilidade, é a de que este *bit* de I/O foi usado no passado, com alguma outra finalidade.

Na preparação para a interrupção (área em cinza), note que a última instrução (EINT) é a que ativa a habilitação geral. Apesar de aqui não ser necessário, seguimos a recomendação do fabricante e precedemos essa instrução com um NOP. Antes do retorno (RET) desta sub-rotina de recepção, apagamos a habilitação geral (DINT). Novamente, como o fabricante recomenda, ela foi seguida por um NOP.

Aqui fazemos um comentário sobre o controle da *flag* GIE. No caso do exemplo, a sub-rotina foi responsável por ativar GIE, assim, no retorno, ele foi desativado. Isso não precisa acontecer. Caso o programador deseje, a *flag* GIE pode permanecer ativada. O mais usual é que o programa principal controle a *flag* GIE e não as sub-rotinas.

Proposta de uma sub-rotina para recepção dos dados por interrupção, de acordo com a Figura 4.1

```
; Subrotina -- REC -- para receber dados
; Inicia com: R5 = Quantidade de dados
;             R6 = Endereço inicial para armazenar os dados

REC:  MOV.B       #0,&P1DIR         ;Porta P1 como entrada
      MOV.B       #0xFF,&P1REN      ;Habilitar resistores de P1
      MOV.B       #0xFF,&P1OUT      ;Selecionar Pullup

      BIC.B       #BIT0,&P2DIR      ;Pino P2.0 como entrada
      BIS.B       #BIT0,&P2REN      ;Habilitar resistor
      BIS.B       #BIT0,&P2OUT      ;Selecionar pullup
      BIC.B       #BIT0,&P2IFG      ;Apagar possível pedido anterior
      BIS.B       #BIT0,&P2IE       ;Habilita interrupção em P2.0
      BIS.B       #BIT0,&P2IES      ;Seleciona flanco de descida em P2.0
      NOP
      EINT                          ;(GIE=1) Habilita as interrupções
```

```
LB1:  CALL        #ROT1             ;Chamar ROT1
      CALL        #ROT2             ;Chamar ROT2
      TEST        R5                ;R5=0?
      JNZ         LB1               ;Repetir ser R6 diferente de zero
      DINT                          ;(GIE=0) Desabilitar as interrupções
      NOP
      RET

; Manipulador (handler) para as interrupções da porta P2
P2_HND:
      ADD         &P2IV,PC          ;Somar deslocamento ao PC
      RETI                          ;P2IV=0, sem interrupção, retorna
      JMP         P2_0_HND          ;P2IV=2, P2.0 interrompeu, saltar
      JMP         P2_1_HND          ;P2IV=4, P2.1 interrompeu, saltar
      JMP         P2_2_HND          ;P2IV=6, P2.2 interrompeu, saltar
      JMP         P2_3_HND          ;P2IV=8, P2.3 interrompeu, saltar
      JMP         P2_4_HND          ;P2IV=10, P2.4 interrompeu, saltar
      JMP         P2_5_HND          ;P2IV=12, P2.5 interrompeu, saltar
      JMP         P2_6_HND          ;P2IV=14, P2.6 interrompeu, saltar
      JMP         P2_7_HND          ;P2IV=16, P2.7 interrompeu, saltar

P2_0_HND:                           ;ISR para interrupção de P2.0
      MOV.B       &P1IN,0(R6)       ;Armazenar dado
      INC.W       R6                ;Incrementar ponteiro de dados
      DEC.W       R5                ;Decrementar contador
      RETI                          ;Retorno para o programa principal

P2_1_HND:               ;ISR para interrupção de P2.1
      ...               ;Aqui começa a rotina
      RETI              ;Retorno para o programa principal

P2_2_HND:               ;ISR para interrupção de P2.2
      ...               ;Aqui começa a rotina
      RETI              ;Retorno para o programa principal

P2_3_HND:               ;ISR para interrupção de P2.3
      ...               ;Aqui começa a rotina
      RETI              ;Retorno para o programa principal

P2_4_HND:               ;ISR para interrupção de P2.4
      ...               ;Aqui começa a rotina
      RETI              ;Retorno para o programa principal

P2_5_HND:               ;ISR para interrupção de P2.5
      ...               ;Aqui começa a rotina
      RETI              ;Retorno para o programa principal

P2_6_HND:               ;ISR para interrupção de P2.6
      ...               ;Aqui começa a rotina
```

```
      RETI              ;Retorno para o programa principal

P2_7_HND:               ;ISR para interrupção de P2.7
      ...               ;Aqui começa a rotina
      RETI              ;Retorno para o programa principal

;-------------------------------------------------------------------------
; Interrupt Vectors
;-------------------------------------------------------------------------
            .sect   ".reset"      ; MSP430 RESET Vector
            .short  RESET
            .sect   ".int42"      ;Posição 42 da Tabela
            .short  P2_HND        ;Endereço a ser colocado na posição 42
```

Vamos detalhar outros dois pontos muito importantes. O primeiro responde à pergunta: como colocar o endereço de uma sub-rotina em uma determinada posição da Tabela de Vetores de Interrupção? Na listagem, o trecho do programa sombreado em verde realiza esta tarefa. De acordo com a Tabela 4.1, a interrupção da porta P2 é a de número 42. Assim:

- `.sect ".int42"` → seleciona a entrada 42 da tabela e
- `.short P2_HND` → armazena o endereço da sub-rotina na posição previamente selecionada. A diretiva `.short` força o endereço da sub-rotina para 16 *bits* e garante o alinhamento em endereço par. Isto é apenas uma garantia.

O leitor curioso pode dar uma olhada no arquivo **msp430f5529.h**. Ele traz as definições que usamos com frequência. Este arquivo costuma estar na pasta:
C:\ti\ccs930\ccs\ccs_base\msp430\include.
(ver .intvec, pag 109 do SLAU 131M)

Também é possível indicar o preenchimento de uma posição da tabela com as duas linhas abaixo. A constante PORT2_VECTOR e as demais estão listadas na tabela disponível no final deste capítulo.

```
      .sect     PORT2_VECTOR
      .short    P2_HND
```

Na listagem acima, o trecho marcado em abóbora é fundamental para o entendimento das interrupções no MSP430. Vimos que a porta P2 compartilha o vetor 42 entre 8 possíveis candidatos e a rotina que atende a essa interrupção precisa descobrir qual desses 8 candidatos, gerou o pedido. Para resolver este caso de forma eficiente, existe o registrador P2IV. Note que a primeira instrução da rotina de interrupção (`ADD &P2IV,PC`) simplesmente soma o valor deste registrador no Contador de Programa. Sabemos que o Contador de Programa sempre aponta para a próxima instrução. Assim, quando a instrução `ADD &P2IV,PC` é executada, PC está apontando para uma instrução RETI. Se não houver interrupção pendente, o registrador P2IV retorna zero e a execução retorna

para o programa principal. Entretanto, se houver alguma interrupção pendente, o registrador P2IV vai retornar um número par que somado ao PC, provoca a seleção de um `jmp` (salto) dentre a sequência de 8 `jmp's` seguidos. Cada um desses `jump's` desvia para uma subrotina diferente. O valor retornado por P2IV é par porque cada `jump` precisa de 2 *bytes*.

Deve ser lembrado que o registrador P2IV sempre retorna o número correspondente à interrupção de maior prioridade pendente e, em seguida, apaga o pedido desta interrupção. Assim, o uso correto desses Registradores de Vetores de Interrupção facilita muito o trabalho do programador.

Mais um detalhe. O leitor pode achar que a sub-rotina P2_ISR está exagerada, pois temos apenas uma única interrupção habilitada. Sim, é verdade. A sub-rotina foi construída desta forma para ilustrar o uso do Registrador Vetor de Interrupção. Se o programa do leitor é simples e há a certeza de que somente a interrupção de P2.0 foi habilitada, então a sub-rotina que atende à esta interrupção pode ser muito mais simples. A listagem abaixo apresenta esta solução. Note que apagamos diretamente o *bit* que indica o pedido de interrupção por P2.0.

```
; Manipulador (handler) para as interrupções da porta P2
P2_HND:                             ;ISR para interrupção de P2.0
      BIC.B       #BIT0,&P2IFG      ;Apagar o pedido de interrupção
      MOV.B       &P1IN,0(R6)       ;Armazenar dado
      INC.W       R6                ;Incrementar ponteiro de dados
      DEC.W       R5                ;Decrementar contador
      RETI                          ;Retorno para o programa principal
```

Nesta outra solução, logo abaixo, fizemos um acesso inofensivo ao registrador P2IV só para que ele fosse lido e com isso apagasse o pedido de interrupção por P2IV. Salvamos o conteúdo de R6 na pilha (`PUSH R6`), copiamos o conteúdo de P2IV para R6 (`MOV &P2IV,R6`) e depois recuperamos o valor original de R6 (`POP R6`).

```
; Manipulador (handler) para as interrupções da porta P2
P2_HND:                             ;ISR para interrupção de P2.0
      PUSH        R6                ;Salvar R6
      MOV         &P2IV,R6          ;Leitura de P2IV]
      POP         R6                ;Restaurar R6
      MOV.B       &P1IN,0(R6)       ;Armazenar dado
      INC.W       R6                ;Incrementar ponteiro de dados
      DEC.W       R5                ;Decrementar contador
      RETI                          ;Retorno para o programa principal
```

4.7. Latência de uma Interrupção

Em geral, os programadores iniciantes têm o conceito equivocado de que a interrupção é algo muito rápido. Isso não é verdade. Se for necessário monitorar um sinal de alta velocidade, é melhor usar o *polling*. A interrupção é interessante pela flexibilidade que traz ao programa.

No MSP430, a interrupção tem uma latência de pelo menos 11 períodos:

- 6 períodos (MCLK) para iniciar a execução da primeira instrução da sub-rotina de interrupção e
- 5 períodos (MCLK) após o `RETI` para continuar com a próxima instrução do programa principal.

A figura abaixo ilustra essas ideias. Na ilustração do lado esquerdo, como a interrupção estava desabilitada, o pedido de interrupção "INTa" não teve qualquer efeito e o programa principal seguiu seu fluxo normal. Já a porção da direita apresenta o caso com interrupção habilitada e, por isso, assim que o processador termina a execução da instrução ele dá início à sequência de eventos que leva ao desvio para rotina que atende à interrupção (ISRa). Note que são 6 períodos de MCLK para iniciar a execução da primeira instrução da ISRa e, no retorno, são 5 períodos de MCLK após o `RETI`, para reassumir o fluxo do programa principal.

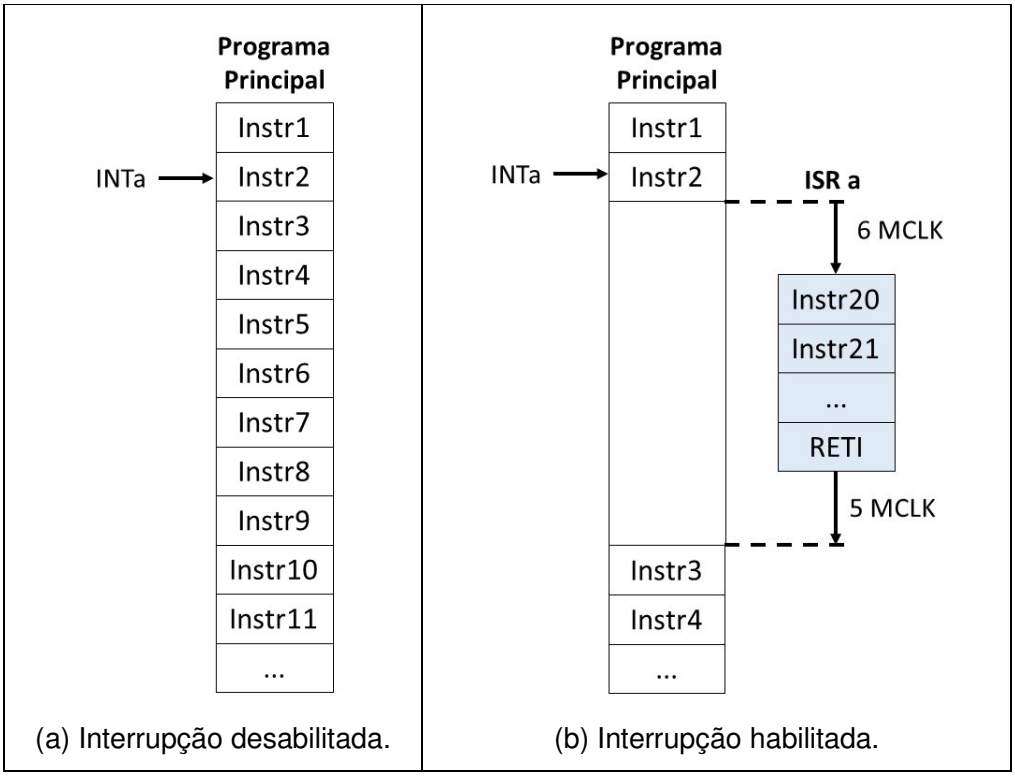

Figura 4.12. Ação de um pedido de interrupção sobre o fluxo de um programa.

A Figura 4.13 ilustra um caso mais complexo. O pedido INTa interrompeu o fluxo do programa principal e provocou a execução de sua ISRa. Entretanto, durante a execução desta ISRa, chegou o pedido INTb. Como uma interrupção não interrompe outra interrupção, este pedido ficou pendente. Assim que a CPU termina a ISRa, ela dá início ao ciclo de ações para desviar para a ISRb. Note que agora não temos uma ideia clara de quanto tempo vai se passar entre o momento em que INTb surgiu e o momento em que ISRb iniciou a execução. Quanto mais longa for ISRa, mais atrasada ficará ISRb. Na melhor das hipóteses, a latência na atenção à ISRb será de 11 períodos de MCLK.

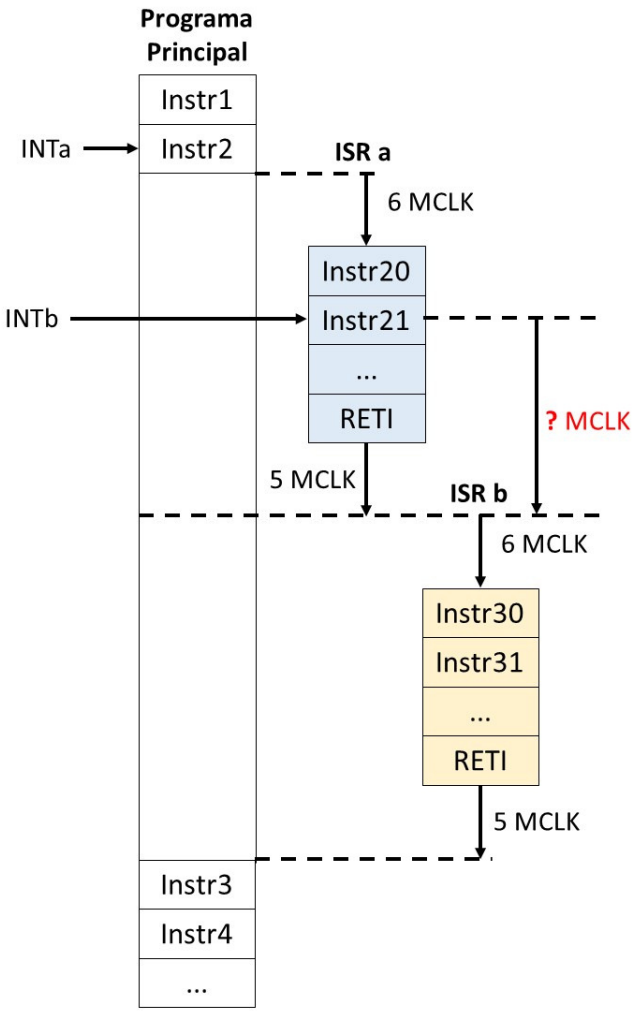

Figura 4.13. Ação das interrupções, quando um pedido de interrupção chega durante a execução da sub-rotina que já estava atendendo a outro pedido.

Essa situação se torna mais complexa quando se tem uma grande quantidade de interrupções habilitadas, o que é comum em sistemas de média ou alta complexidade. Isso tudo, justifica a seguinte regra de ouro das interrupções.

As rotinas que atendem às interrupções devem ser as mais curtas possíveis.

Nunca "trave" uma ISR aguardando um evento externo! Por exemplo, é suicídio prender a execução dentro de uma rotina de interrupção só para esperar o usuário acionar uma chave. Tampouco use laços com grande quantidade de repetições. Por exemplo, se dentro da ISR se decidir que é necessário enviar 100 *bytes* pela porta serial, não o faça "dentro" da sub-rotina de interrupção. Programe o recurso de interrupção da porta serial e faça essa transmissão usando 100 vezes essa outra interrupção.

O programador deve ter em mente que o recurso CPU é único e que ele é disputado por diversas interrupções. Por isso, cada rotina de interrupção deve ser bem comportada. Ao ganhar o recurso CPU, a ISR deve usá-lo da forma mais rápida possível, de forma que todas as demais interrupções tenham chance. É preciso lembrar de que os pedidos não atendidos ficam pendentes, aguardando sua oportunidade. O processador, ao analisar as pendências dá atenção aos mais prioritários (maior número dentro da Tabela de Vetores de Interrupção). Existe aqui um outro perigo: pode ser que algumas interrupções de alta prioridade aconteçam numa taxa tão alta a ponto de impedir que as de baixa prioridade tenham qualquer acesso à CPU.

Mais um esclarecimento se faz necessário. Cada interrupção tem origem numa determinada *flag* que, como já afirmamos, nesta arquitetura tem seu nome finalizando com as letras "IFG". Vamos considerar o caso em que uma interrupção está em execução e uma outra qualquer "pede" 3 vezes por sua interrupção. Quando for atender à essa outra interrupção, a CPU não tem como saber que foram 3 pedidos. O primeiro pedido, ativa a *flag* correspondente e os demais pedidos, como a *flag* já está ativa, não têm qualquer efeito prático.

4.8. O *Pipeline* do MSP e suas Interrupções

Aqui neste tópico, vamos explicar o porquê da recomendação de se usar `NOPs` junto com as instruções `EINT` e `DINT`. Este tópico é um pouco mais aprofundado e o leitor pode pulá-lo de acordo com seu interesse.
(Verificar se está correto, não encontrei explicação detalhada confirmando as ideias aqui colocadas).

Já vimos no Capítulo 1 que o MSP430 faz uso de um *pipeline* de 3 estágios:

- (B) Busca: que acessa a memória e traz a próxima instrução a ser executada;
- (D) Decodifica: que "entende" o que faz a instrução e prepara sua execução e
- (E) Executa: que executa a instrução.

Como mostrado na figura abaixo, têm-se 3 ações simultâneas: uma instrução sendo executada em "X", a próxima sendo decodificada em "D" e uma terceira sendo buscada por "B". É importante que fique claro que a instrução só é realmente executada quando passa pela unidade "X". Assim, examinando o *pipeline* é óbvio que a execução da **Instr_x** pode interferir na busca pela **Instr_z**, mas não tem qualquer ingerência sobre **Instr_y**, que já está na unidade de decodificação. Agora vem a pergunta que será respondida no próximo parágrafo: o que acontece se **Instr_x** for `EINT` ou `DINT`?

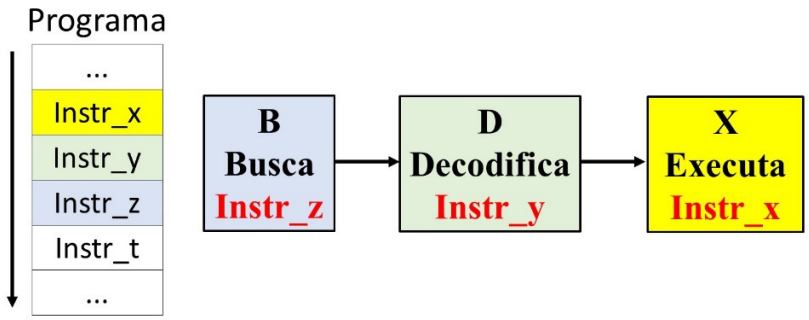

Figura 4.14. Ilustração do pipeline do MSP430 em operação.

O manual afirma que a próxima instrução após o `EINT` sempre é executada, mesmo que haja alguma interrupção pendente. Isto está claro na Figura 4.15. A instrução **Instr_y** já está no *pipeline* e não há alternativa senão executá-la. Na verdade, a outra solução seria invalidar o *pipeline* e refazê-lo novamente, mas isto é proibitivo, pois consumiria muito tempo.

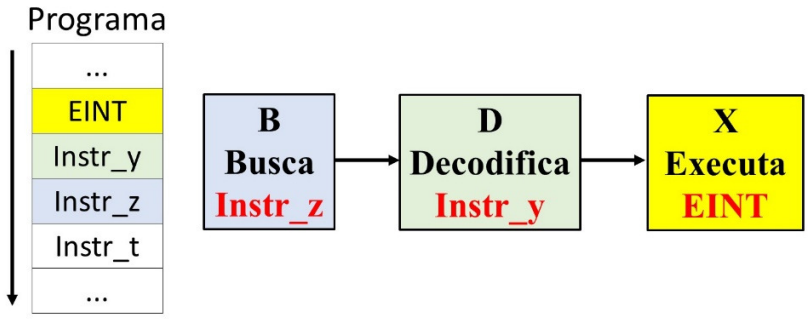

Figura 4.15. Pipeline do MSP430 quando a instrução EINT é executada.

Abordemos agora uma situação um pouco mais complexa, que está apresentada na Figura 4.16. Vamos imaginar que há um pedido de interrupção pendente que foi causado por uma alteração em P2.0. Assim, o *bit* 0 do registrador P2IFG está em 1 (P2IFG.0 = 1). O programador decide ativar a habilitação geral (`EINT`) e em seguida apagar o pedido de interrupção pendente (esta instrução está representado na figura simplesmente por P2IFG = 0). Como foi afirmado que após `EINT` a próxima instrução é sempre executada, então esta instrução que apaga o pedido é executada e a interrupção que estava pendente não deve acontecer. Entretanto, após a instrução `EINT` a CPU já iniciou as ações para aceitar a interrupção por P2.0. Por isso, no próximo ciclo do *pipeline* ela é surpreendida pela instrução que apaga o pedido. O que será que vai acontecer? O fabricante não dá detalhes sobre esse caso. Não podemos afirmar com certeza se a interrupção vai acontecer ou não (meta estabilidade). Convidamos o leitor a escrever um programa para testar este caso.

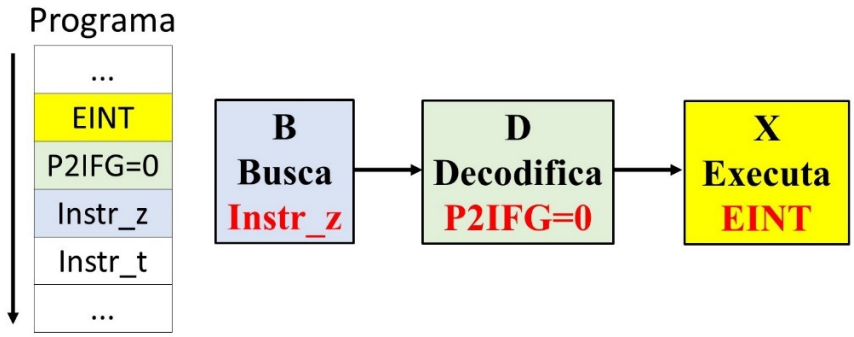

Figura 4.16. Pipeline do MSP430 quando a instrução EINT é seguida por uma instrução que apaga um pedido de interrupção que estava pendente.

Entretanto, se logo após a instrução `EINT`, colocamos um `NOP`, como recomenda o fabricante, isso não acontece, pois a instrução que apaga o pedido de interrupção ainda vai estar na fase de busca. Veja a figura abaixo. Assim evitamos qualquer tipo de incerteza. O montador do CCS vai além e exige que a instrução `EINT` seja protegida por dois `NOPs`, sendo um antes e outro depois. Isso também vale para a instrução `DINT`. Um `NOP` antes e um `NOP` depois evitam que o programador use as instruções `EINT` e `DINT`, uma em seguida à outra. Uma condição análoga à essa no *pipeline* explica o porquê de o montador gerar um alerta pedindo a introdução de um NOP após algumas instruções de salto. Quando se programa em C, o compilador se encarrega de todos esses cuidados.

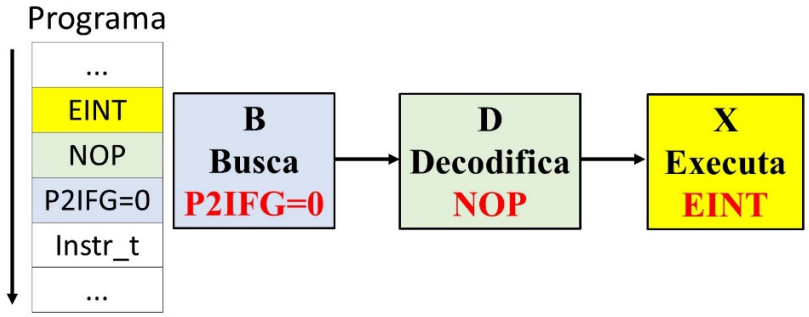

Figura 4.17. Pipeline do MSP430 quando a instrução EINT é sucedida por um NOP, o que evita incerteza com a instrução que apaga o pedido de interrupção que estava pendente.

4.9. Exercícios Resolvidos

Apresentamos a seguir alguns exercícios resolvidos envolvendo interrupções. De certa forma eles são limitados porque, por enquanto, só sabemos gerar interrupções com as portas P1 e P2. Mas essas duas portas serão suficientes para trabalharmos os principais conceitos. À medida que prosseguimos com os demais capítulos, estudaremos outros recursos do MSP430 e abordaremos as demais interrupções.

ER 4.1. Use a chave S2 (P1.1) para comandar um pino de I/O que vai provocar a interrupção por flanco de descida no pino P2.0. Para que se possa observar essa interrupção vamos inverter o estado do LED1. Num exercício mais adiante, usaremos uma das chaves para provocar interrupções.

Solução:
Para este exercício vamos usar uma lógica que pode parecer excessiva ao leitor. Usando um pequeno cabo, vamos colocar em curto os pinos P1.5 e P2.0. Configuramos o pino P2.0 para gerar uma interrupção a cada flanco de descida e usamos o pino P1.5, controlado pela chave S2, para provocar este flanco.

Em resumo, cada vez que a chave S2 for acionada, o pino P1.5 vai para zero, o que resulta num flanco de descida em P2.0 e gera a interrupção que inverte o estado do LED1. Preferimos fazer desta forma para que o pulso que gera a interrupção seja bem "limpo" e assim não cause surpresas.

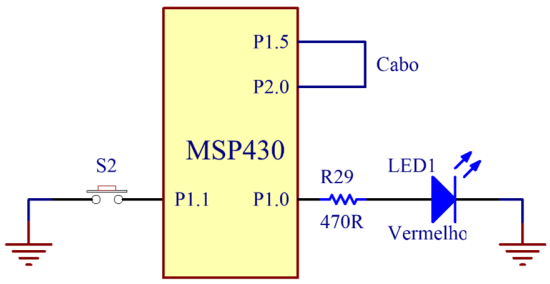

Figura 4.18. Detalhamento da solução do ER 4.1. O leitor apenas precisa usar um cabo para fechar o curto entre P1.5 e P2.0.

Antes de partirmos para a solução, vamos apresentar um programa intermediário que, sem usar interrupção, inverte o estado do LED1 a cada acionamento de S2. No capítulo anterior estudamos diversos programas deste tipo. O atual é muito semelhante ao ER 3.7. Nosso programa, após configurar as portas, fica preso num laço aguardando o usuário acionar S2. Quando S2 é acionada, o estado do LED1 é invertido e o programa entra num outro laço esperando o usuário liberar a chave S2 e, então, tudo se repete. Esta é a solução que denominamos *polling*, estamos fazendo o *polling* da chave S2. Para evitar acionamentos indevidos no controle de S2, usamos a sub-rotina *debounce*.

ER 4.1a: Listagem de uma solução que usa apenas polling (não usa interrupção)

```
; ER 4.01a
; Chave S2 inverte estado de LED2

DBC:        .equ        1000            ;Atraso para debounce

ER04p01a:   CALL        #IO_CONFIG
LB1:        BIT.B       #BIT1,&P1IN     ;Esperar acionar S2
            JNZ         LB1
            XOR         #BIT0,&P1OUT    ;Inverter estado LED2
            CALL        #DEBOUNCE
LB2:        BIT.B       #BIT1,&P1IN     ;Esperar soltar S2
            JZ          LB2
            CALL        #DEBOUNCE
            JMP         LB1
;
IO_CONFIG:  BIC.B       #BIT1,&P1DIR    ;P1.1 = entrada com Pullup
            BIS.B       #BIT1,&P1REN
            BIS.B       #BIT1,&P1OUT
            ;
            BIS.B       #BIT0,&P1DIR    ;Led vermelho
            BIC.B       #BIT0,&P1OUT    ;Apagado
            RET
;
```

```
; Atraso para debounce
DEBOUNCE:   MOV         #DBC,R5
DB1:        DEC         R5
            JNZ         DB1
            RET
```

Vamos apresentar logo a seguir a listagem da solução que faz uso de interrupção. Nesta solução é interessante ver o trecho que inicializa a porta P2 e que habilita a interrupção por flanco de descida em P2.0. A estrutura é muito semelhante ao programa anterior. Porém, cada vez que S2 é acionada, o pino P1.5 vai para zero. Quando S2 é liberada, P1.5 volta para um. A passagem de P1.5 de nível alto para baixo gera o flanco de descida necessário para provocar uma interrupção por P2.0.

Na listagem, a sub-rotina `P2_ISR` que atende à interrupção por P2.0 está marcada em abóbora. Essa sub-rotina é muito simples: apaga a *flag* em P2IFG, inverte o estado de LED1 e retorna (`RETI`). Como temos a certeza de que somente uma interrupção por P2 está habilitada, não precisamos descobrir quem causou a interrupção. Podemos então omitir a consulta a P2IV. Entretanto, é necessário apagar a *flag* correspondente no registrador P2IFG. Fazemos isso diretamente com a instrução `BIC` ou com uma leitura do registrador P2IV. Lembre-se de que a leitura de P2IV apaga a *flag* da interrupção de maior prioridade.

Ao final do programa está o trecho que armazena na posição 42 da Tabela de Vetores de Interrupção o endereço da sub-rotina `P2_ISR`. O preenchimento desta tabela precisa ser feito junto com a carga do programa, pois ela está localizada na memória *Flash* do MSP430 e o programa não tem como alterá-la durante a execução.

Não vamos estudar esta possibilidade, mas já vimos que existe a opção de indicar ao processador para usar a Tabela de Vetores de Interrupção a partir do topo da memória SRAM. Neste caso, os programas poderiam alterar os vetores durante a execução.

ER 4.1b: Listagem da solução usando interrupção

```
; ER 4.01b
; Fechar curto entre P2.0 e P1.5
; Chave S2 provoca flanco descida em P1.5
; P1.5 provoca interrupção em P2.0

DBC:        .equ        1000            ;Atraso para debounce

ER04p01a:   CALL        #IO_CONFIG
            NOP
            EINT                        ;GIE=1, Habilitação Geral
            NOP
LB1:        BIT.B       #BIT1,&P1IN     ;Esperar acionar S2
            JNZ         LB1
```

```
            CALL        #DEBOUNCE
            BIC.B       #BIT5,P1OUT       ;P1.5=0 --> flanco de descida
            ;
LB2:        BIT.B       #BIT1,&P1IN       ;Esperar soltar S2
            JZ          LB2
            CALL        #DEBOUNCE
            BIS.B       #BIT5,P1OUT       ;P1.5=1
            JMP         LB1
;
IO_CONFIG:  BIC.B       #BIT1,&P1DIR      ;P1.1 = entrada com Pullup
            BIS.B       #BIT1,&P1REN
            BIS.B       #BIT1,&P1OUT
            ;
            BIS.B       #BIT0,&P1DIR      ;Led vermelho
            BIC.B       #BIT0,&P1OUT      ;Apagado
            ;
            BIS.B       #BIT5,&P1DIR      ;P1.5 --> P2.0
            BIS.B       #BIT5,&P1OUT      ;P1.5 aciona interrup em P2.0
            ;
            BIC.B       #BIT0,&P2DIR      ;P2.0 = entrada com Pullup
            BIS.B       #BIT0,&P2REN
            BIS.B       #BIT0,&P2OUT
            BIC.B       #BIT0,&P2IFG      ;Apagar possível pendência
            BIS.B       #BIT0,&P2IES      ;Flanco de descida
            BIS.B       #BIT0,&P2IE       ;Habilita interrupção
            RET
;
; Debounce de S2
DEBOUNCE:   MOV         #DBC,R5
DB1:        DEC         R5
            JNZ         DB1
            RET
;
; Rotina de interrupção para P2.0
P2_ISR:     BIC.B       #BIT0,P2IFG
            XOR.B       #BIT0,&P1OUT      ;Inverter estado de LED1
            RETI

;-------------------------------------------------------------------------
; Stack Pointer definition
;-------------------------------------------------------------------------
            .global __STACK_END
            .sect   .stack

;-------------------------------------------------------------------------
; Interrupt Vectors
;-------------------------------------------------------------------------
            .sect   ".reset"        ; MSP430 RESET Vector
            .short  RESET
```

```
            .sect ".int42"          ;Vetor da porta P2
            .short  P2_ISR          ;Endereço a ser armazenado
```

Novamente, vamos dar atenção à maneira do programa indicar ao montador os endereços das sub-rotinas a serem colocadas nas diversas posições da Tabela de Vetores de Interrupção. Uma consulta à Tabela 4.1, ou à tabela detalhada que está no final deste capítulo, permite ver que a Porta P2 usa o vetor 42. O trecho abaixo é uma repetição do que já vimos.

- `.sect ".int42"` → seleciona a entrada 42 da tabela e
- `.short P2_HND` → fornece o endereço a ser armazenado na posição indicada.

Também se consegue o mesmo resultado com as duas linhas abaixo

```
            .sect PORT2_VECTOR      ;Vetor da porta P2
            .short  P2_ISR          ;Endereço a ser armazenado
```

Novamente, convidamos o leitor para dar uma olhada no arquivo **msp430f5529.h**. Ele costuma estar na pasta: C:\ti\ccs930\ccs\ccs_base\msp430\include.

ER 4.2. O leitor que ao longo do capítulo anterior sofreu com o gerenciamento das chaves, deve achar tentador usar interrupções para gerenciá-las. Será que vai simplificar? Vamos então a um problema simples que pede para usar as interrupções das chaves para inverter o estado dos *leds* e contar seus acionamentos, da seguinte forma:

- Chave S1 (P2.1) → inverte o estado de LED1 (P1.0) e incrementa R5.
- Chave S2 (P1.1) → inverte o estado de LED2 (P4.7) e incrementa R6.

Solução:
As chaves S1 e S2 estão nas duas portas passíveis de operar com interrupções. Vamos então programá-las para interromper com o flanco de descida, pois as chaves, quando acionadas, fecham curto para a terra. Assim, cada vez que uma das interrupções acontecer, simplesmente invertemos o *led* correspondente. Antecipando problemas, usamos registradores para contar os acionamentos. Por enquanto, nada dissemos sobre rebotes. O que esperar deles?

O programa apresentado na listagem a seguir é simples. Ele inicia chamando a sub-rotina que inicializa as portas e os *leds*. É interessante ver com cuidado o trecho que prepara a interrupção para cada uma das chaves. Depois, o programa faz a habilitação geral (`EINT`) e fica esperando pelas interrupções acontecerem. As sub-rotinas que atendem as duas interrupções estão sombreadas em abóbora. Elas são elementares. Apagam o pedido, invertem o *led* e incrementam o contador. Ao final da listagem, sombreado em verde, está o trecho que indica ao carregador (*loader*) o que colocar nos vetores 42 e 47.

ER 4.2: Listagem de uma solução usando interrupção

```
; ER 4.2
; Usando interrupções
; S1 (P2.1) inverte led vermelho (P1.0)
; S2 (P1.1) inverte led verde (P4.7)

ER04p02a:   CALL        #IO_CONFIG
            CLR         R5          ;Contar acionamentos de S1
            CLR         R6          ;Contar acionamentos de S2
            NOP
            EINT                    ;GIE=1, Habilitação Geral
            NOP
            JMP         $           ;Esperar interrupções

; (P2.1) S1 Interrupção
P2_ISR:     BIC.B       #BIT1,&P2IFG      ;Apagar pedido de interrupção
            XOR.B       #BIT0,&P1OUT      ;Inverter led vermelho (P1.0)
            INC         R5                ;Incrementar contador
            RETI

; (P1.1) S2 Interrupção
P1_ISR:     BIC.B       #BIT1,&P1IFG      ;Apagar pedido de interrupção
            XOR.B       #BIT7,&P4OUT      ;Inverter led verde (P4.7)
            INC         R6                ;Incrementar contador
            RETI

IO_CONFIG:  BIC.B       #BIT1,&P2DIR      ;S1 = P2.1 = entrada com Pullup
            BIS.B       #BIT1,&P2REN
            BIS.B       #BIT1,&P2OUT
            BIC.B       #BIT1,&P2IFG      ;Apagar possível pendência
            BIS.B       #BIT1,&P2IES      ;Selecionar Flanco de descida
            BIS.B       #BIT1,&P2IE       ;Habilita interrupção
            ;
            BIC.B       #BIT1,&P1DIR      ;S2 = P1.1 = entrada com Pullup
            BIS.B       #BIT1,&P1REN
            BIS.B       #BIT1,&P1OUT
            BIC.B       #BIT1,&P1IFG      ;Apagar possível pendência
            BIS.B       #BIT1,&P1IES      ;Selecionar Flanco de descida
            BIS.B       #BIT1,&P1IE       ;Habilita interrupção
            ;
            BIS.B       #BIT0,&P1DIR      ;P1.0 = Led vermelho
            BIC.B       #BIT0,&P1OUT      ;Apagado
            ;
            BIS.B       #BIT7,&P4DIR      ;P4.7 = Led vermelho
            BIC.B       #BIT7,&P4OUT      ;Apagado
            RET

;-------------------------------------------------------------------------
; Stack Pointer definition
;-------------------------------------------------------------------------
```

```
            .global __STACK_END
            .sect   .stack

;-------------------------------------------------------------------------
; Interrupt Vectors
;-------------------------------------------------------------------------
            .sect   ".reset"              ; MSP430 RESET Vector
            .short  RESET

            .sect   ".int42"              ;P2 vetor
            .short  P2_ISR

            .sect   ".int47"              ;P1 vetor
            .short  P1_ISR
```

Ao rodar o programa, a primeira impressão é a de que tudo está perfeito e que conseguimos uma boa solução para o problema de monitorar as chaves. Entretanto, isto não é verdade. Após alguns acionamentos o leitor vai poder constatar que o rebotes trazem alguma perturbação. Com os contadores em zero, experimente, usando simultaneamente dois dedos, acionar 5 vezes as duas chaves e confira os valores de R5 e R6. Também podem surgir problemas ao soltar a chave, para isso, experimente reter a chave pressionada e depois, observando o *led*, libere-a. Faça isso algumas vezes. O *led* mudou de estado?

O leitor deve ter se convencido de que os rebotes perturbam nossa rotina e que precisam ser tratados. O problema é: como tratar rebotes nas interrupções? Para os rebotes que surgem com o acionamento de uma chave, podemos colocar um pequeno atraso ao entrar na sub-rotina de interrupção. Não é uma boa solução, mas pode resolver. O grande problema são os rebotes ao liberar a chave, já que não sabemos quanto tempo o usuário a mantém pressionada. Ficar “dentro” da rotina de interrupção esperando a liberação da chave não é solução, pois isso vai impedir todas as demais interrupções.

Os rebotes não se fazem tão evidentes porque a interrupção consome tempo e ameniza o problema. Além disso, o relógio atual da placa deve ser aproximadamente de 1 MHz (1.048.576 Hz). Se elevarmos a frequência deste relógio, aumenta o efeito nefasto dos rebotes.

Conclusão: não use interrupção para monitorar as chaves!

ER 4.3. A exemplo do problema que iniciou este capítulo, vamos propor um caso de transmissão de dados. Para não termos de fazer uma grande quantidade de conexões, vamos trabalhar com a transmissão serial *bit-a-bit* com um sinal de sincronismo (STRB). A cada acionamento da chave S1, deverá ser transmitida a palavra “FEBEAPA”, cujo final está marcado com um *byte* igual a zero. O esquema está detalhado na figura abaixo.

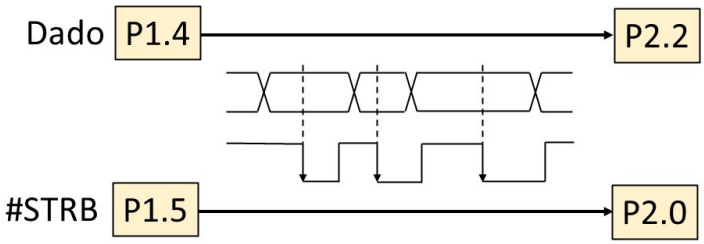

Figura 4.19. Proposta de um enlace para comunicação serial síncrona. A Launch Pad transmite e ela própria recebe.

O protocolo indicado é simples. O pino P1.5 indica com um flanco de descida cada novo *bit* que é colocado em P1.4. Usando a interrupção em P2.0, vamos receber os *bits* e montar a palavra e guardar na memória.

Solução:
Vamos criar um laço principal que fica aguardando o acionamento da chave S1. Quando isso acontece, a palavra é transmitida e o programa volta a esperar o acionamento de S1 para fazer uma nova transmissão. Vamos usar os seguintes registradores

Tabela 4.3. Emprego dos registradores

Transmissor	Receptor
R5 = ponteiro para leitura	R10 = ponteiro para escrita
R6 = contador de *bits* (8, 7, ..., 0)	R11 = contador de *bits* (8, 7, ..., 0)
R7 = *byte* a ser transmitido	R12 = montar os *bits*

A cada acionamento de S1, o programa principal chama a rotina ROT_TX que *bit*-a-*bit* transmite em sequência os *bytes* apontados por R5, até encontrar o zero, que marca o final da palavra. Cada *byte* tem seu *bit* menos significativo transmitido primeiro. Para cada *bit* que é colocado em P1.4, é gerado um pulso em nível baixo em P1.5.

A cada interrupção em P2.0, o *bit* P1.4 é copiado para R12 que vai montando o *byte*. Cada *byte* montado é escrito na memória usando o ponteiro R10. O leitor, que já estudou o Capítulo 3 (Assembly), não terá dificuldades em entender a lógica de transmissão. Usando o recurso de "Memory Browser" em 0x2400, o leitor vai poder observar a frase original e a frase recebida. Para melhorar a percepção, os *leds* são acionados. O *led* vermelho, durante a transmissão e o *led* verde durante a recepção. É claro que eles acendem ao mesmo tempo. Como tudo é muito rápido, o leitor vai ver uma rápida piscada.

ER 4.3: Listagem da solução para transmissão e recepção serial usando o mesmo MSP

```
; ER 4.3
; Transmissão serial
; STRB = P1.5 ----> P2.0
; Dado = P1.4 ----> P2.2

DBC:        .equ  1000              ;Atraso para debounce
BIT210:     .equ  BIT2|BIT1|BIT0    ;0000 0111 = 7
BIT54:      .equ  BIT5|BIT4         ;0011 0000 = 0x30
;
; Registradores empregados
; R5  = ponteiro origem
; R6  = contador bits origem
; R7  = separar bits
; R10 = ponteiro destino
; R11 = contador bits destino
; R12 = montar bits
;
; Programa principal
ER04p03:    CALL  #IO_CONFIG
            NOP
            EINT                    ;GIE=1, Habilitação Geral
            NOP
LB0:        MOV   #STR_TX,R5        ;Ponteiro TX
            MOV   #STR_RX,R10       ;Ponteiro RX
            MOV   #8,R11            ;Inicializar contador bits interrup
LB1:        BIT.B #BIT1,&P2IN       ;S1 acionada?
            JNZ   LB1
            CALL  #ROT_TX           ;Chamar rotina que transmite string
            CALL  #DEBOUNCE
LB2:        BIT.B #BIT1,&P2IN       ;S1 liberada?
            JZ    LB2
            CALL  #DEBOUNCE
            JMP   LB0
;
; Rotina para transmissão serial, finaliza no ZERO final
ROT_TX:     BIS.B #BIT0,P1OUT       ;Led vermelho aceso
LB_TX0:     MOV.B @R5+,R7           ;Ler próximo byte
            TST.B R7                ;Próximo = zero
            JNZ   LB_TX1            ;Diferente de Zero, continua
            BIC.B #BIT0,P1OUT       ;Led vermelho apagado
            RET                     ;Terminou
LB_TX1:     MOV   #8,R6             ;Contar 8 bits
LB_TX2:     RRC   R7                ;Carry = LSbit
            BIS.B #BIT4,&P1OUT      ;P1.4 = 1
            JC    LB_TX3            ;Carry = 1?
            BIC.B #BIT4,&P1OUT      ;P1.4 = 0, pois Carry = 0
LB_TX3:     BIC.B #BIT5,&P1OUT      ;P1.5 = STRB = 0
            BIS.B #BIT5,&P1OUT      ;P1.5 = STRB = 1
            DEC   R6                ;Decrementa contador
```

```
            JNZ   LB_TX2            ;Se cont != 0, mais um bit
            JMP   LB_TX0            ;Se cont == 0, acabou o byte
;
; ISR: Sub-rotina para atender interrupção por P2.0
P2_ISR:     BIS.B #BIT7,&P4OUT      ;Acender led Verde
            BIC.B #BIT0,&P2IV       ;Apagar pedido interrup
            BIT.B #BIT2,P2IN        ;Carry = bit que chegou
            RRC.B R12               ;Deslocar para direita com Carry
            DEC   R11               ;Chegaram 8 bits?
            JZ    P2_LB1            ;Sim, pula
            RETI                    ;Caso negativo, retorna
P2_LB1:     MOV.B R12,0(R10)        ;Gravar na memória
            INC   R10               ;Avançar ponteiro
            MOV.B #8,R11            ;Contador pronto para
            BIC.B #BIT7,P4OUT       ;Apagar led Verde
            RETI
;
; Sub-rotina para configurar I/O
IO_CONFIG:  BIC.B #BIT210,&P2DIR    ;P2.2=P2.1=P2.0 = entrada com Pullup
            BIS.B #BIT210,&P2REN
            BIS.B #BIT210,&P2OUT
            BIC.B #BIT0,&P2IFG      ;P2.0 - Apagar possível pendência
            BIS.B #BIT0,&P2IES      ;P2.0 - Selecionar Flanco de descida
            BIS.B #BIT0,&P2IE       ;P2.0 - Habilita interrupção
                  ;
            BIS.B #BIT54|BIT0,&P1DIR      ;P1.5=P1.4=P1.0 = Saída
            BIS.B #BIT54,&P1OUT     ;P1.4 = P1.5 = 1
            BIC.B #BIT0,&P1OUT      ;P1.0=LED1=apagado
;
            BIS.B #BIT7,&P4DIR      ;P4.7 = Saída
            BIC.B #BIT7,&P4OUT      ;P4.7=LED2=apagado
            RET
;
; Atraso para debounce
DEBOUNCE:   MOV   #DBC,R5
DB1:        DEC   R5
            JNZ   DB1
            RET
;
; Segmento de dados inicializados
            .data
STR_TX:     .byte "FEBEAPA",0       ;Sequência a ser transmitida
STR_RX:     .byte "xxxxxxx",0       ;Espaço para receber sequência

;-------------------------------------------------------------------------
; Stack Pointer definition
;-------------------------------------------------------------------------
            .global __STACK_END
            .sect   .stack
```

```
;-------------------------------------------------------------------------
; Interrupt Vectors
;-------------------------------------------------------------------------
            .sect   ".reset"                ; MSP430 RESET Vector
            .short  RESET

            .sect   ".int42"                ; Vetor Interrupção P2
            .short  P2_ISR
```

Curiosidade: Quem está sendo homenageado neste exercício?

ER 4.4. O exercício anterior foi feito usando apenas uma LaunchPad. Um único MSP transmitia e recebia o que havia transmitido. Isto até parece marmelada. Vamos agora testar o mesmo problema, mas usando placas distintas, como mostrado na figura. A cada acionamento de da chave S1, palavra é transmitida.

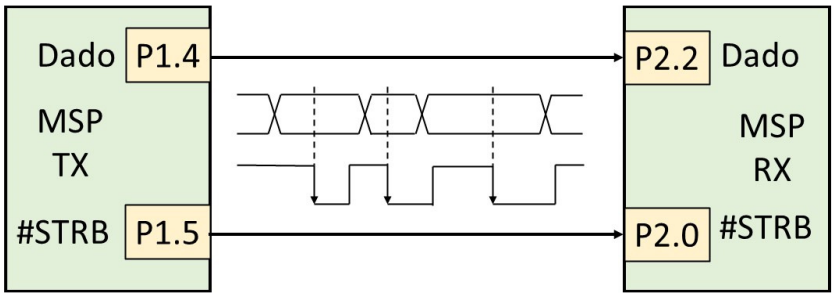

Figura 4.20. Proposta de um enlace para comunicação serial síncrona entre dois LaunchPads.

Solução:
A solução será exatamente a mesma, mas quebrada em duas frações, uma para cada MSP, como mostrado nas listagens abaixo. Antes de qualquer coisa, fazemos as duas conexões indicadas na figura acima. Note que não precisamos fazer a conexão de terra entre as duas placas porque ela é feita através do USB. Se o leitor estiver usando computadores diferentes, um para cada placa, pode ser necessária a conexão de terra entre elas.

Essa solução é um pouco trabalhosa pois temos agora de ver como usar dois MSPs e o CCS ao mesmo tempo. Existem duas possibilidades, mas antes conecte os pinos indicados (P1.5 com P2.0 e P1.4 com P2.2):

- Crie duas instâncias (inicie duas vezes) o CCS, cada uma usando uma área de trabalho diferente. Assim, eles (os CCS) ficam completamente separados e é possível comandar (*debug*) as duas placas de forma independente.
- Use apenas uma instância do CCS. Para isso conecte o MSP TX e grave nele o programa transmissor. Saia do modo *debug*, remova do USB o MSP TX e agora

conecte o MSP RX. Carregue nele o programa de recepção e entre no modo *debug*. Só agora volte a conectar o USB do MSP TX, que inicia executando o programa transmissor previamente carregado. Por segurança, acione seu botão de *reset*. Depois disso tudo, rode o programa de recepção usando o CCS. Acione S1 na placa transmissora e examine o resultado da recepção.

ER 4.4 TX: Listagem da solução para a transmissão serial

```
; ER 4.4 TX
; Transmissor serial
; STRB = P1.5 ----> P2.0
; Dado = P1.4 ----> P2.2

DBC:        .equ  1000              ;Atraso para debounce
BIT54:      .equ  BIT5|BIT4         ;0011 0000 = 0x30

; R5  = ponteiro origem
; R6  = contador bits origem
; R7  = separar bits
;
ER04p04tx:  CALL  #IO_CONFIG
LB0:        MOV   #STR_TX,R5        ;Ponteiro TX
LB1:        BIT.B #BIT1,&P2IN       ;S1 acionada?
            JNZ   LB1
            CALL  #ROT_TX           ;Chamar rotina que transmite string
            CALL  #DEBOUNCE
LB2:        BIT.B #BIT1,&P2IN       ;S1 liberada?
            JZ    LB2
            CALL  #DEBOUNCE
            JMP   LB0
;
; Rotina para transmissão serial, finaliza no ZERO final
ROT_TX:     BIS.B #BIT0,P1OUT       ;Led vermelho aceso
LB_TX0:     MOV.B @R5+,R7           ;Ler próximo byte
            TST.B R7                ;Próximo = zero
            JNZ   LB_TX1            ;Diferente de Zero, continua
            BIC.B #BIT0,P1OUT       ;Led vermelho apagado
            RET                     ;Terminou
LB_TX1:     MOV   #8,R6             ;Contar 8 bits
LB_TX2:     RRC   R7                ;Carry = LSbit
            BIS.B #BIT4,&P1OUT      ;P1.4 = 1
            JC    LB_TX3            ;Carry = 1?
            BIC.B #BIT4,&P1OUT      ;P1.4 = 0, pois Carry = 0
LB_TX3:     BIC.B #BIT5,&P1OUT      ;P1.5 = STRB = 0
            ;NOP
            ;NOP
            ;NOP
            ;NOP
            ;NOP
```

```
            ;NOP
            ;NOP
            BIS.B #BIT5,&P1OUT      ;P1.5 = STRB = 1
            DEC   R6                ;Decrementa contador
            JNZ   LB_TX2            ;Se cont != 0, mais um bit
            JMP   LB_TX0            ;Se cont == 0, acabou o byte
;
; Sub-rotina para configurar I/O
IO_CONFIG:  BIC.B #BIT1,&P2DIR      ;S1 = P2.1 = entrada com Pullup
            BIS.B #BIT1,&P2REN
            BIS.B #BIT1,&P2OUT
            ;
            BIS.B #BIT54|BIT0,&P1DIR  ;P1.5=P1.4=P1.0 = Saída
            BIS.B #BIT54,&P1OUT     ;P1.5=P1.4=1
            BIC.B #BIT0,&P1OUT      ;P1.0=LED1=apagado
            RET
;
; Atraso para debounce
DEBOUNCE:   MOV   #DBC,R5
DB1:        DEC   R5
            JNZ   DB1
            RET
;
; Gravar sequência na Flash
STR_TX:     .byte "FEBEAPA",0       ;Sequência a ser transmitida
```

ER 4.4 RX: Listagem da solução para a recepção serial

```
; ER 4.4 RX
; Receptor serial
; STRB = P1.5 ----> P2.0
; Dado = P1.4 ----> P2.2

BIT20:      .equ  BIT2|BIT0   ;0000 0101 = 5

; R10 = ponteiro destino
; R11 = contador bits destino
; R12 = montar bits
;
ER04p03:    CALL  #IO_CONFIG
            MOV   #STR_RX,R10
            MOV   #8,R11
            NOP
            EINT                    ;GIE=1, Habilitação Geral
            NOP
            JMP   $
;
; ISR: Sub-rotina para atender interrupção por P2.0
```

```
P2_ISR:     BIS.B #BIT7,&P4OUT      ;Acender led Verde
            BIC.B #BIT0,&P2IV       ;Apagar pedido interrup
            BIT.B #BIT2,P2IN        ;Carry = bit que chegou
            RRC.B R12               ;Deslocar para direita com Carry
            DEC   R11               ;Chegaram 8 bits?
            JZ    P2_LB1            ;Sim, pula
            RETI                    ;Caso negativo, retorna
P2_LB1:     MOV.B R12,0(R10)        ;Gravar na memória
            INC   R10               ;Avançar ponteiro
            MOV.B #8,R11            ;Contador pronto para
            BIC.B #BIT7,P4OUT       ;Apagar led Verde
            RETI
;
; Sub-rotina para configurar I/O
IO_CONFIG:  BIC.B #BIT20,&P2DIR     ;P2.2=P2.1=P2.0 = entrada com Pullup
            BIS.B #BIT20,&P2REN
            BIS.B #BIT20,&P2OUT
            BIC.B #BIT0,&P2IFG      ;P2.0 - Apagar possível pendência
            BIS.B #BIT0,&P2IES      ;P2.0 - Selecionar Flanco de descida
            BIS.B #BIT0,&P2IE       ;P2.0 - Habilita interrupção
                  ;
            BIS.B #BIT7,&P4DIR      ;P4.7 = Saída
            BIC.B #BIT7,&P4OUT      ;P4.7=LED2=apagado
            RET
;
; Segmento de dados inicializados
            .data
STR_RX:     .byte "xxxxxxx",0       ;Espaço para receber sequência

;-------------------------------------------------------------------------
; Stack Pointer definition
;-------------------------------------------------------------------------
            .global __STACK_END
            .sect   .stack

;-------------------------------------------------------------------------
; Interrupt Vectors
;-------------------------------------------------------------------------
            .sect   ".reset"                ; MSP430 RESET Vector
            .short  RESET

            .sect   ".int42"                ; Vector porta P2
            .short  P2_ISR
```

As duas listagens são simples e de fácil entendimento. Note que no programa transmissor, a palavra a ser transmitida foi armazenada na memória *Flash*, pois não usamos o indicador "`.data`". Isso é necessário pois há a possibilidade do MSP TX ser desconectado do USB. Neste caso ele perde a alimentação e, é claro, todo o conteúdo de sua RAM. Ao ser reconectado, o conteúdo da RAM é imprevisível. O programa de

recepção, após a configuração, fica preso em um laço infinito aguardando as interrupções acontecerem.

Agora, vamos ao conceito que motiva este exercício. Ao rodar os programas, tal qual estão nas listagens, o leitor vai constatar que a recepção falha. Entretanto, se "descomentar" (remover os pontos-e-vírgulas) a série de `NOPs` marcados em amarelo na rotina transmissora, a recepção funciona sem erros. Por que será que os `NOPs` são necessários, se no exercício anterior não ocorreu erro algum.

O leitor já deve desconfiar da resposta: a taxa de transmissão está além da capacidade da recepção. A inserção dos `NOPs` diminui a taxa de transmissão e assim a recepção acontece normalmente. Qual será a quantidade mínima de `NOPs`?

No exercício anterior, estávamos usando o mesmo processador, assim, o flanco de descida em P1.5 provocava a interrupção que era imediatamente aceita e interrompia a transmissão. Com dois processadores separados, isto não mais acontece.

ER 4.5. Neste exercício vamos ver o caso em que são usadas várias interrupções da porta P1 de forma que passa a ser necessário consultar o Registrador de Vetores de Interrupção (P1IV) para saber quem causou a interrupção. Como, por enquanto, temos apenas as portas para gerar interrupções, vamos propor um experimento que não é muito elegante, mas é funcional. Vamos conectar cabos a 5 pinos da porta P1, como mostrado na figura abaixo. Esses 5 pinos terão suas interrupções habilitadas para flanco de descida, assim, cada vez que "tocarmos" um deles à terra, uma interrupção acontece. Cada interrupção realiza uma ação nos *leds*, como abaixo especificado.

- P1.2 → apaga o *led* vermelho (P1.0);
- P1.3 → acende o *led* vermelho (P1.0);
- P1.4 → apaga o *led* verde (P4.7);
- P1.5 → acende o *led* verde (P4.7) e
- P1.6 → inverte o estado dos dois *leds*.

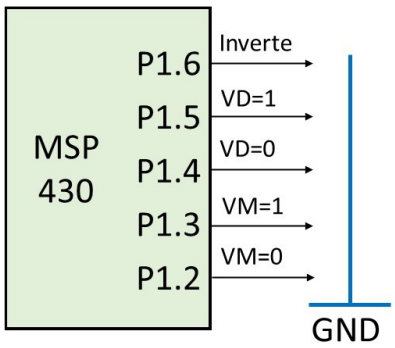

Figura 4.21. Uso de 5 pinos de I/O para controlar os leds por interrupção.

Solução:
A solução é muito simples e está apresentada abaixo. Basicamente, o programa faz a inicialização das portas de I/O, habilita o *flag* geral (GIE = 1) e fica para num laço infinito esperando as interrupções acontecerem. Cada vez que acontece uma interrupção, a sub-rotina P1_ISR é executada. Note a lógica com P1IV para descobrir quem provocou a interrupção. Para as interrupções que não foram habilitadas, colocamos (por segurança) apenas um `RETI`. Se fossemos otimistas poderíamos imaginar que elas nunca ocorreriam e poderíamos omitir essa parte. Porém, como se costuma dizer, um otimista nunca chega vivo até a idade da aposentadoria. Os demais desvios simplesmente acendem, apagam ou invertem os *leds*.

O leitor pode estar se perguntando sobre os rebotes. É claro que a forma como estamos fazendo o contato com a terra gera uma grande quantidade de rebotes. Porém, como a interrupção só faz uma única coisa, não importa se ela acontece uma vez ou várias vezes. Isto não é verdade para a interrupção por P1.6, que inverte os *leds*. Com esta interrupção o leitor vai perceber claramente a interferência dos rebotes. Uma forma interessante de constatar a presença de rebotes é colocar um contador independente para cada interrupção.

ER 4.5: Listagem da solução para controlar os leds via interrupção da porta P1

```
ER 4.5RX
; Controlar leds por interrupção
;
; Vermelho: P1.2 --> apaga e P1.3 --> acende
; Verde:    P1.4 --> apaga e P1.5 --> acende
; Inverter: P1.6 (inverte ambos os leds)

BITS        .equ  BIT6|BIT5|BIT4|BIT3|BIT2      ;0111 1100 = 0x7C
;
; Programa principal
ER04p05:    CALL  #IO_CONFIG
            NOP
            EINT                    ;GIE=1
            NOP
            JMP   $                 ;Laço infinito
;
; Interrupção da porta P1
P1_ISR:     ADD         &P1IV,PC    ;Somar deslocamento ao PC
            RETI                    ;P1IV=0, sem interrupção, retorna
            JMP         P1_0_HND    ;P1IV=2, P1.0 interrompeu, saltar
            JMP         P1_1_HND    ;P1IV=4, P1.1 interrompeu, saltar
            JMP         P1_2_HND    ;P1IV=6, P1.2 interrompeu, saltar
            JMP         P1_3_HND    ;P1IV=8, P1.3 interrompeu, saltar
            JMP         P1_4_HND    ;P1IV=10, P1.4 interrompeu, saltar
```

```
            JMP         P1_5_HND    ;P1IV=12, P1.5 interrompeu, saltar
            JMP         P1_6_HND    ;P1IV=14, P1.6 interrompeu, saltar
            JMP         P1_7_HND    ;P1IV=16, P1.7 interrompeu, saltar

P1_0_HND:                           ;Sem ação, retornar
P1_1_HND:                           ;Sem ação, retornar
P1_7_HND:   RETI                    ;Sem ação, retornar

P1_2_HND:   BIC.B #BIT0,P1OUT       ;Vermelho, apagar
            RETI

P1_3_HND:   BIS.B #BIT0,P1OUT       ;Vermelho, acender
            RETI

P1_4_HND:   BIC.B #BIT7,P4OUT       ;Verde, apagar
            RETI

P1_5_HND:   BIS.B #BIT7,P4OUT       ;Verde, acender
            RETI

P1_6_HND:   XOR.B #BIT0,P1OUT       ;Vermelho, inverter
            XOR.B #BIT7,P4OUT       ;Verde, inverter
            RETI
;
; Sub-rotina para configurar I/O
IO_CONFIG:  BIS.B #BIT0,&P1DIR      ;Vermelho P1.0 = Saída
            BIC.B #BIT0,&P1OUT      ;LED1=apagado
            ;
            BIS.B #BIT7,&P4DIR      ;verde P4.7 = Saída
            BIC.B #BIT7,&P4OUT      ;LED2=apagado
            ;
            BIC.B #BITS,P1DIR       ;Bits como entrada
            BIS.B #BITS,P1REN
            BIS.B #BITS,P1OUT       ;Bits com pullup
            BIC.B #BITS,&P1IFG      ;Apagar possíveis pendências
            BIS.B #BITS,&P1IES      ;Selecionar Flanco de descida
            BIS.B #BITS,&P1IE       ;Habilita interrupções
            RET

;-------------------------------------------------------------------------
; Stack Pointer definition
;-------------------------------------------------------------------------
            .global __STACK_END
            .sect   .stack

;-------------------------------------------------------------------------
; Interrupt Vectors
;-------------------------------------------------------------------------
            .sect   ".reset"        ; MSP430 RESET Vector
            .short  RESET
```

```
            .sect   ".int47"        ; Vector Porta P1
            .short  P1_ISR
```

4.10. Exercícios Propostos

A seguir são propostos alguns exercícios. Novamente, afirmamos que como temos apenas as portas P1 e P2 para gerar interrupções, os exercícios propostos são de certa forma limitados ou repetitivos.

EP 4.1. Vamos propor um jogo de disputa e usar interrupções para ver qual das chaves, S1 ou S2, é acionada primeiro. Se S1 for acionada primeiro, o acenda o *led* vermelho. Caso seja a chave S2, acenda o *led* verde. O jogo recomeça quando as duas chaves estiverem soltas. Será que os rebotes interferem em sua solução?

EP 4.2. Ainda considerando o jogo de acionamento das chaves S1 ou S2, tente uma abordagem desabilitando as interrupções. A primeira interrupção que acontecer, desabilita as demais e escreve algo num registrador qualquer. Esta solução, se for bem empregada, pode evitar os rebotes.

EP 4.3. Agora, vamos tornar o programa um pouco mais desafiante. Com o jogo de disputa no acionamento das chaves S1 ou S2, transfira para o programa principal a tarefa de acender o *led* da chave vencedora. As duas sub-rotinas só podem agir sobre um único registrador, que será consultado pelo programa principal. Não se pode desabilitar as interrupções. Então, como evitar os rebotes?
Dica: tente inventar uma operação (uma conta com duas etapas) que dê resultados diferentes em função da ordem em que é executada.

EP 4.4. A figura abaixo apresenta uma boa solução para se eliminar os rebores de uma chave mecânica. O pequeno capacitor de cerâmica funciona como um curto para os rebotes que têm alta frequência. Usando uma chave extra e um capacitor, faça a montagem indicada na figura e realize experimentos para verificar se os rebotes foram realmente eliminados ou apenas reduzidos.

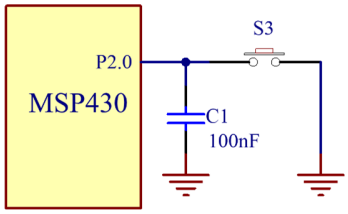

Figura 4.22. Uso de um capacitor para diminuir os rebotes gerados pelo acionamento de uma chave.

EP 4.5. A Figura 4.23 apresenta duas soluções para se detectar a interrupção de um feixe luminoso com um foto-transistor. O TIL 78 é um foto-transistor infravermelho, mas possui também uma boa sensibilidade na faixa de luz visível. A base deste transistor está exposta, assim, a luz incidente provoca a geração de cargas portadoras e o transistor funciona como um curto. Quando não há luz, não há portadores e o transistor está cortado. Note o funcionamento semelhante ao de uma chave. O encapsulamento do TIL 78 é transparente e o pino maior corresponde ao emissor. A sugestão da direita faz uso de um inversor Schmitt-Trigger TTL (74LS14), que garante transições mais abruptas (as entradas do MSP já têm seu próprio Schmitt-Trigger) e o resistor R1 funciona como controle de sensibilidade, quanto maior o valor de R1, mais sensível fica o sensor. Em suma, ele opera da seguinte forma:

- Com luz → nível baixo e
- Sem luz → nível alto.

Monte um dos esquemas sugeridos e usando os dois *leds* como um contador binário, conte o número de vezes que o sensor é iluminado, ou sombreado.

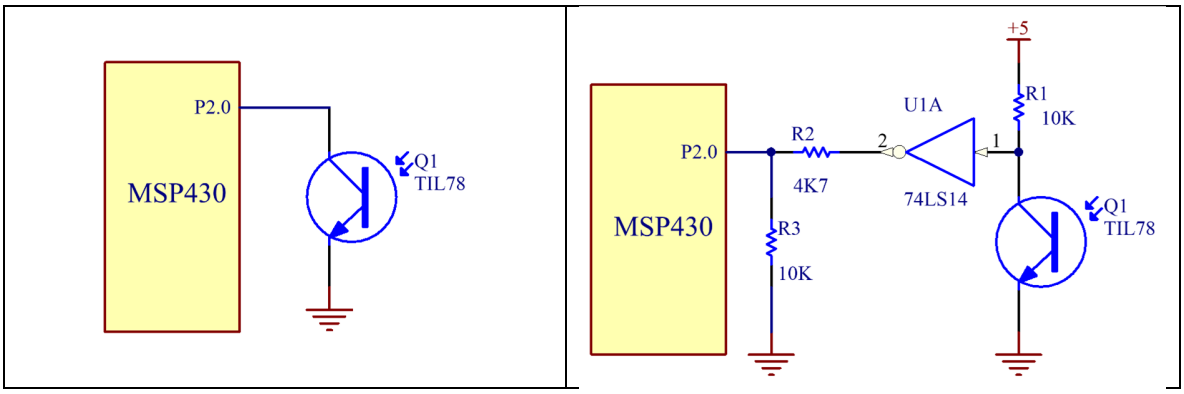

Figura 4.23. Sugestão de emprego do foto-transistor infravermelho (TIL 78) como um sensor de luz.

EP 4.6. Nesta proposta, vamos controlar os *leds* com um estímulo sonoro. Uma batida de palmas, um grito ou qualquer outro som que o leitor imagine. Para tanto, vamos usar um microfone de eletreto. O sinal na saída de tais microfones é muito pequeno e precisa de um amplificador com um ganho na faixa de 250 a 300 vezes. Esse ganho, é claro, depende do volume sonoro entregue ao microfone.

A Figura 4.24 apresenta a proposta de um pré-amplificador muito simples para um microfone de eletreto. Foi usado o TL072, mas existem alternativas. O amplificador está

conectado entre o +3,3V e a terra. Por isso, é usado um divisor resistivo (R2 e R3) para gerar uma tensão intermediária que alimenta a entrada positiva. O capacitor C2 é usado para eliminar ruído. O resistor R1 polariza o microfone e o capacitor C1 elimina o nível DC do sinal, ou seja, deixa passar apenas sua componente alternada. Os resistores R4, R5 e R6 controlam o ganho e, portanto, o nível de sinal entregue ao pino. O ganho é dado pela equação abaixo.

$$G = -\frac{R5 + R6}{R4}$$

O resistor R5 garante o ganho mínimo de – 220 vezes. Com o potenciômetro no máximo, podemos chegar ao ganho de – 330 vezes. O leitor vai precisar fazer o ajuste experimental.

Vamos ao problema. Usando *polling* (e não interrupção) use as chaves S1 e S2 para inverter o estado dos *leds*.

- Chave S1 → cada acionamento inverte o estado do *led* vermelho;
- Chave S2→ cada acionamento inverte o estado do *led* verde e
- Som → inverte o estado dos dois *leds*.

Conecte a saída do amplificador ao pino P2.0 e habilite a interrupção por flanco de subida ou descida (tanto faz). Cada vez que essa interrupção acontecer, ela inverte o estado dos dois *leds*. Use uma batida de palmas para tentar acionar a interrupção. O estímulo sonoro será longo (vários flancos), por isso, assim que acontecer a primeira interrupção, iniba as próximas. Fica para o leitor o desafio de como efetivar a inibição das próximas interrupções.

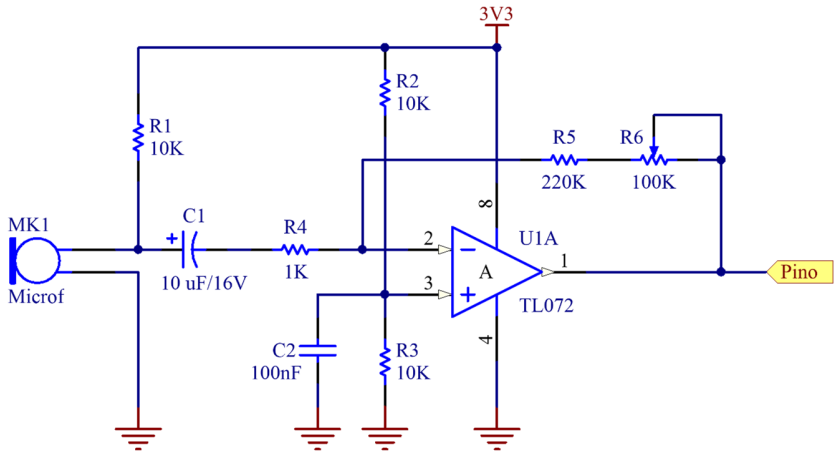

Figura 4.24. Proposta de um pré-amplificador simples para permitir que o som capturado por um microfone de eletreto acione um dos pinos do MSP. Alternativas: TL074, TL082, TL084, LM324, e vários outros.

Tabela 4.4. Detalhamento da Tabela de Vetores de Interrupção do MSP430F5529

Prioridade	Constante	Fonte	*Flags* de interrupção	Reg IV	Endereço
63 (mais alta)	*RESET*_VECTOR	*Reset*	WDTIFG, KEYV	SYSRSTIV	0xFFFE
62	SYSNMI_VECTOR	NMI Sistema	SVMLIFG, SVMHIFG, DLYLIFG, DLYHIFG, VLRLIFG, VLRHIFG, VMAIFG, JMBNIFG, JMBOUTIFG	SYSSNIV	0xFFFC
61	UNMI_VECTOR	NMI Usuário	NMIIFG, OFIFG, ACCVIFG, BUSIFG	SYSUNIV	0xFFFA
60	COMPB_VECTOR	Comparador B	*Flags* de interrupção	CBIV	0xFFF8
59	TIMER_B0_VECTOR	Timer TB0	TB0CCR0.CCIFG	-	0xFFF6
58	TIMER_B1_VECTOR	Timer TB0	TB0CCR[1,2,3,4,5,6].CCIFG TB0CR0.TB0IFG	TB0IV	0xFFF4
57	WDT_VECTOR	Watchdog Timer	WDTIFG	-	0xFFF2
56	USCI_A0_VECTOR	USCI A0	UCA0RXIFG, UCA0TXIFG	UCA0IV	0xFFF0
55	USCI_B0_VECTOR	USCI B0	UCB0RXIFG, UCB0TXIFG	UCB0IV	0xFFEE
54	ADC12_VECTOR	Conversor AD	ADC12IFG0,1,2, …, 15	ADC12IV	0xFFEC
53	TIMER0_A0_VECTOR	Timer TA0	TA0CCR0 CCIFG0	-	0xFFEA
52	TIMER0_A1_VECTOR	Timer TA0	TA0CCR[1,2,3,4].CCIFG TA0CR0.TA0IFG	TA0IV	0xFFE8
51	USB_UBM_VECTOR	Porta USB	*Flags* de interrupção	USBIV	0xFFE6
50	DMA_VECTOR	DMA	DMA0IFG, DMA1IFG, DMA2IFG	DMAIV	0xFFE4
49	TIMER1_A0_VECTOR	Timer TA1	TA1CCR0.CCIFG	-	0xFFE2
48	TIMER1_A1_VECTOR	Timer TA1	TA1CCR[1,2].CCIFG TA1CR0.TA0IFG	TA1IV	0xFFE0
47	PORT1_VECTOR	Porta P1	P1IFG.0,1, …, 7	P1IV	0xFFDE
46	USCI_A1_VECTOR	USCI A1	UCA1RXIFG, UCA1TXIFG	UCA1IV	0xFFDC

45	USCI_B1_VECTOR	USCI B1	UCB1RXIFG, UCB1TXIFG	UCB1IV	0xFFDA
44	TIMER2_A0_VECTOR	Timer TA2	TA2CCR0.CCIFG	-	0xFFD8
43	TIMER2_A1_VECTOR	Timer TA2	TA2CCR[1,2].CCIFG TA2CR0.TA0IFG	TA2IV	0xFFD6
42	PORT2_VECTOR	Porta P2	P2IFG.0, 1, …, 7	P2IV	0xFFD4
41	RTC_VECTOR	Real-Time Clock	RTCRDYIFG, RTCTEVIFG, RTCAIFG, RT0PSIFG, RT1PSIFG	RTCIV	0xFFD2
40	Reservada	Reservada			0xFFD0
...	Reservada	Reservada	...	...	...
0 (mais baixa)	Reservada	Reservada		0XFF80	0XFF80