5 MSP430 e C

Versão 1.0

Este capítulo pretende fazer a ponte entre a programação *assembly* que foi usada até agora e a programação em C, que será empregada deste ponto em diante. Esta transição é simples e intuitiva, razão pela qual este é um capítulo curto. Vamos expor os principais conceitos para programar o MSP430 usando a linguagem C e reescreveremos a solução de alguns exercícios resolvidos dos capítulos anteriores. O fato de se programar em uma linguagem de alto nível traz grandes benefícios, pois passamos a ter recursos sofisticados para estruturas de dados, melhor controle de fluxo de programa e as bibliotecas nos oferecem uma enormidade de recursos. Além disso tudo, é claro, temos programas mais inteligíveis.

Entretanto, o *assembly* não deve ser menosprezado. Ele é fundamental para o programador interpretar corretamente seus programas e ter segurança no que está fazendo. Além disso, algumas vezes se faz necessário examinar o *assembly* gerado pelo compilador para entendermos algum comportamento inesperado ou para decidir a melhor forma de se realizar uma operação.

Para os leitores que não têm experiência em programação em C ou para aqueles que estão “enferrujados”, o Apêndice C oferece uma breve revisão.

5.0. Quero Programar o MSP430 Usando C e Não Pretendo Ler Todo Este Capítulo

Neste caso, comece imediatamente a programar e aprenda por demanda. Use sua intuição, pois a forma de programação é bem comportada e intuitiva. Todos os registradores dos dispositivos de I/O podem ser acessados usando seus nomes em letras

maiúsculas. As constantes BIT0, BIT1 etc, já estão definidas. É recomendado apenas que veja os detalhes para uso de interrupções que estão apresentados no item 5.4. Entretanto, é fortemente recomendado o estudo deste capítulo.

5.1. Acesso aos Registradores

O título deste tópico deveria ser “Constantes Previamente Definidas”, porém isso dificultaria a identificação para quem fizesse a busca pelo Sumário, por isso usamos o título acima.

O compilador C do CCS traz previamente definido os endereços de todos os registradores dos dispositivos de I/O do MSP430. O leitor mais curioso pode dar uma olhada no arquivo **msp430f5529.h**. Ele traz uma grande quantidade de constantes. Este arquivo costuma estar na pasta:
C:\ti\ccs930\ccs\ccs_base\msp430\include.

Apresentamos abaixo alguns exemplos:

```
// Constantes para manipular bits
#define BIT0      (0x0001)
#define BIT1      (0x0002)
...
#define BITE      (0x4000)
#define BITF      (0x8000)

// Bits do registrador de Status
#define C         (0x0001)
#define Z         (0x0002)
#define N         (0x0004)
#define V         (0x0100)
#define GIE       (0x0008)

// Não é exatamente isto que está no arquivo msp430f5529.h
// Apresentamos o resultado final
#define P1IN      (0x200)
#define P2IN      (0x201)
#define P1OUT     (0x202)
...
```

Todos os registradores podem ser acessados diretamente, como se fossem uma variável qualquer de seu programa. É claro que podem existir registradores que só operam para leitura e outros que só operam para escrita. Nestes casos, que são poucos, é preciso algum cuidado. Por hora, não precisamos nos preocupar com isso. Apresentamos alguns exemplos.

```
int x;

P1DIR = 1;              //P1.0 é saída e P1.1 até P1.7 são entradas
x=P1IN;                 //Leitura dos 8 bits da porta P1
P2OUT = P2OUT | 1;      //Pino P2.0 em nível alto

// Podemos até fazer coisas malucas
P1DIR = P2DIR + 13;
P2OUT = 2*P1IN;
```

5.2. Como Operar com Registradores

Afirmamos que os registradores podem ser operados como se fossem variáveis. Esta afirmação é um pouco exagerada, pois é preciso um cuidado para não se alterar inadvertidamente os demais *bits* do registrador. Por exemplo, se precisarmos programar o pino P1.0 para operar como saída e usarmos o *statement* `P1DIR = 1;` colocamos realmente P1DIR.0 = 1, porém todos os demais *bits* deste registrador vão para zero. Isto significa que todos os demais pinos foram configurados como entrada. Será que é isso que queríamos? Apresentamos então o exemplo abaixo que indica como alterar individualmente um *bit* de um registrador qualquer. Usamos como exemplo o *bit* 5 do registrador P1OUT. É recomendado o estudo da Figura C.4 do Apêndice C.

```
P1OUT = P1OUT |  BIT5;  // Quero P1OUT.5 = 1
P1OUT = P1OUT & ~BIT5;  // Quero P1OUT.5 = 0
P1OUT = P1OUT ^  BIT5;  // Quero inverter P1OUT.5

//Podemos usar uma forma mais compacta, note a operação antes do sinal =
P1OUT |=  BIT5;   // Quero P1OUT.5 = 1
P1OUT &= ~BIT5;   // Quero P1OUT.5 = 0
P1OUT ^=  BIT5;   // Quero inverter P1OUT.5
```

Alguns registradores do MSP430 usam campos de vários *bits* para especificar uma configuração. A figura abaixo apresenta, apenas como exemplo ilustrativo, o registrador de controle do Timer A, que será estudado no Capítulo 7. Notem que diversas configurações são feitas com campos de 2 bits. Se quisermos especificar esse *timer* para operar no Modo Contínuo, teremos de fazer o campo MC = 2. Esse campo MC ocupa os bits 4 e 5, assim, precisamos do bit 5 = 1 e bit 4 = 0. Se quisermos especificar para usar o relógio ACLK, precisamos fazer o campo TASSEL = 1, ou seja, bit 9 = 0 e bit 8 = 1.

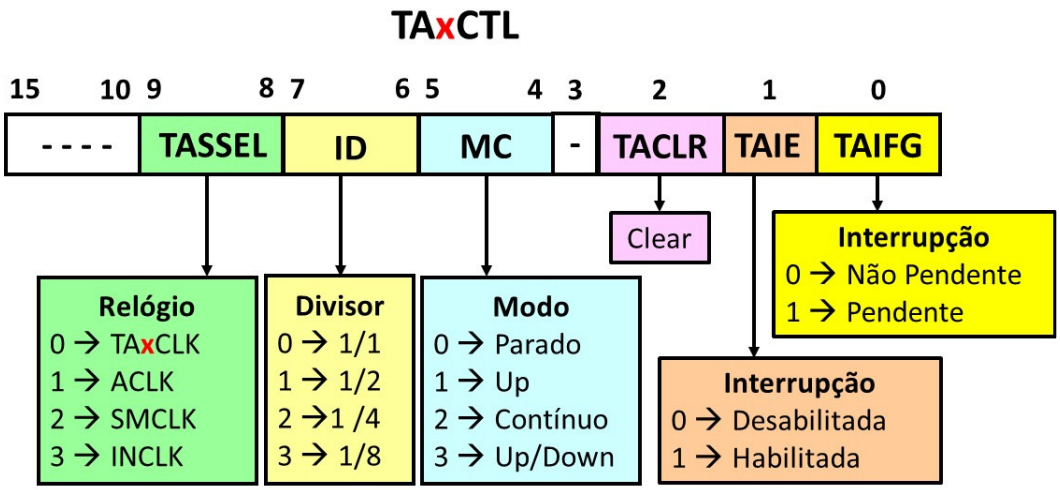

Figura 5.1. Descrição dos bits do registrador TAxCTL (Reset faz TAxCTL=0), que será estudado no Capítulo 7.

Para facilitar operações desse tipo, o arquivo cabeçalho MSP430.h traz as constantes exemplificadas abaixo. Todos os recursos de configuração do MSP430 estão previamente definidos de forma análoga. Elas vão facilitar em muito a configuração desses diversos registradores. Note que os campos MC e TASSEL podem ser especificados de duas formas diferentes.

```
#define TAIFG     BIT0
#define TAIE      BIT1
#define TACLR     BIT2

#define MC_0      0                 //Modo Stop
#define MC_1      (BIT4)            //Modo Up
#define MC_2      (BIT5)            //Modo Continuous
#define MC_3      (BIT5 | BIT4)     //Modo up/down

#define MC_STOP         0                 //Modo Stop
#define MC_UP           (BIT4)            //Modo Up
#define MC_CONTINUOUS   (BIT5)            //Modo Continuous
#define MC_UPDOWN       (BIT5 | BIT4)     //Modo up/down

#define ID_0      0
#define ID_1      (BIT6)
#define ID_2      (BIT7)
#define ID_3      (BIT7 | BIT6)

#define TASSEL_0  0             //clk = TACLK
#define TASSEL_1  (BIT8)        //clk = ACLK
#define TASSEL_2  (BIT9)        //clk = SMCLK
#define TASSEL_3  (BIT9 | BIT8) //clk = INCLK
```

```
#define TASSEL_TACLK    0                 //clk = TACLK
#define TASSEL_ACLK     (BIT8)            //clk = ACLK
#define TASSEL_SMCLK    (BIT9)            //clk = SMCLK
#define TASSEL_INCLK    (BIT9 | BIT8)     //clk = INCLK
```

Assim, de forma bem simples conseguimos configurar os diversos campos deste registrador. Veja os exemplos a seguir.

```
TAxCTL |= TAIE;          //bit TAIE=1 (habilita a interrupção)
TAxCTL |= TASSEL_1;      //Selecionar ACLK
TAxCTL |= MC_2;          //Operar no modo 2
```

É claro que tudo pode ser combinado em uma única linha. Digamos que se quer ativar o *clear* (TCLR = 1), a interrupção (TAIE = 1), selecionar o modo 2 (MC = 2), programar o divisor por 8 (ID = 3) e usar o relógio SMCLK (TASSEL = 2). A linha abaixo faz isso tudo. Note que foi usada a atribuição, veja o sinal de igualdade.

```
TAxCTL = TASSEL_2 | ID_3 | MC_2 | TACLR | TAIE;
```

Um erro muito comum é se esquecer de que a operação OR, sinalizada pela barra vertical, apenas consegue ativar bits e nunca *zerá-los*. Veja abaixo o exemplo equivocado onde o programador configurou o *timer* para operar no modo 2 (MC = 2) e depois precisou mudar para o modo 1 (MC = 1). No calor da geração de códigos, ele se equivocou e acabou selecionando MC = 3. A seguir estão o procedimento equivocado e o correto.

```
// Configuração inicial do registrador de controle
TAxCTL = TASSEL_2 | ID_3 | MC_2 | TACLR | TAIE;
...
...
// Procedimento errado
// Agora, é necessário mudar para o modo 1 (MC = 1)
TAxCTL |= MC_1;          //Operação equivocada, ativa o modo 3

// Procedimento correto
// Agora, é necessário mudar para o modo 1 (MC = 1)
TAxCTL &= ~MC_3;         //Zerar o campo MC
TAxCTL |= MC_1;          //Operação correta, ativa o modo 1
```

Muitos estudantes se confundem com a operação `TAxCTL &= ~MC_3` . Em binário, temos que `MC_3 = 0000 0000 0011 0000`, então `~MC_3 = 1111 1111 1100 1111`. Note que uma operação AND com este número vai zerar apenas os bits 4 e 5 e manter inalterado os demais.

5.3. GPIO e Controle do Fluxo de Programa em C

Ao programar microcontroladores, fatalmente enfrentamos a situação de ter de reagir a *bits* de I/O. Algumas vezes precisamos aguardar o *bit* ir para algum estado e outras vezes alteramos o fluxo do programa de acordo com o estado de algum *bit* de I/O. Abordaremos este problema de forma prática, resolvendo alguns problemas do Capítulo 3.

ER 5.1 (ER 3.6). Escreva um programa em C que mantenha aceso o LED1 (P1.0), que é o vermelho, enquanto a chave S1 (P2.1) permanecer acionada e o LED2 (P4.7), que é o verde, enquanto a chave S2 (P1.1) permanecer acionada.

Solução:
Precisamos fazer as configurações listadas abaixo.

P1.0 (LED1) → saída	P4.7 (LED2) → saída
P2.1 (S1) → entrada com *pullup*	P1.1(S2) → entrada com *pullup*.

A listagem abaixo apresenta a solução deste exercício. Nesta listagem, colocamos de forma artificial os números das linhas. Por hora, esses números devem ser ignorados e, é claro, não devem ser inseridos no seu programa. Mais adiante precisaremos usar de tais números.

É interessante ler com cuidado a função `io_config()` que configura as duas chaves e os dois *leds*. Nesta função, cada linha em C vai corresponder a uma linha em *assembly*. Veja que o programa principal se resume a um laço infinito, caracterizado por `while(1){...}`.

Observe a terceira linha da listagem do programa, onde é declarado o protótipo da função: `void io_config(void)`. Essa declaração é interessante porque indica para o compilador como é a função, quais argumentos recebe e que tipo de argumento retorna. Ao compilar um programa, quando o compilador encontra uma função desconhecida, é considerado o tipo padrão que é a função que recebe um argumento inteiro e retorna um argumento inteiro. Depois, quando encontra a declaração da função é gerado um alerta (*warning*) caso não corresponda à suposição feita. Por isso, é interessante que se apresente o protótipo de todas as funções logo no início do programa.

Alguns programadores preferem iniciar seu arquivo com todas as funções (na sequência em que são usadas) e deixar para o final a função `main()`, assim, não é necessário indicar o protótipo das funções. É questão de gosto. O leitor faça como achar melhor. No presente texto, considerando a finalidade didática, preferimos apresentar o protótipo de todas as funções logo no início do código.

Para descobrir o estado das chaves S1 e S2, usamos as operações:

- `if ( (P2IN&BIT1) == BIT1) //se chave aberta ...` e
- `if ( (P1IN&BIT1) == 0)    //se chave fechada ...`

Note que simples referência ao nome do registrador já indica sua leitura. Basta então ler o registrador e usar uma operação AND (&) para isolar o *bit* de interesse e fazer o teste correspondente. Para a chave S1, temos:

- Chave S1 aberta: `P2IN & BIT1 = BIT1`
- Chave S1 fechada: `P2IN & BIT1 = 0`

Listagem da solução do ER 5.1

```
1: // ER 5.1 (ER 3.6)
2:
3: #include <msp430.h>
4:
5: void io_config(void);      //Protótipo da função
6:
7: void main(void){
8:    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer
9:    io_config();      //Configurar GPIO
10:   while(1){         //Laço infinito
11:         if ( (P2IN&BIT1) == 0)        P1OUT |= BIT0;    //Fechada
12:         else                          P1OUT &= ~BIT0;   //Aberta
13:         if ( (P1IN&BIT1) == 0)        P4OUT |= BIT7;    //Fechada
14:         else                          P4OUT &= ~BIT7;   //Aberta
15:   }
16:}
17:
18:// Configurar GPIO
19:void io_config(void){
20:   P1DIR |= BIT0;    //Led1 = P1.0 = saída
21:   P1OUT &= ~BIT0;   //Led1 apagado
22:
23:   P4DIR |= BIT7;    //Led2 = P4.7 = saída
24:   P4OUT &= ~BIT7;   //Led1 apagado
25:
26:   P2DIR &= ~BIT1;   //S1 = P2.1 = entrada
27:   P2REN |= BIT1;    //Habilitar resistor
28:   P2OUT |= BIT1;    //Habilitar pullup
29:
30:   P1DIR &= ~BIT1;   //S2 = P2.2 = entrada
31:   P1REN |= BIT1;    //Habilitar resistor
32:   P1OUT |= BIT1;    //Habilitar pullup
33:}
```

Acho que todo programador que já praticou *assembly* e que migra para C deve ser perguntar: que raios de código *assembly* meu programa em C resultou? Para satisfazer essa curiosidade, apresentamos abaixo a listagem do *assembly* que o compilador do CCS gerou para o programa acima. Note que no comentário que é colocado em cada linha, o

número entre duas barras verticais (`|n|`) indica a correspondência com a linha de programa em C.

É importante comentar que o endereçamento das portas fez uso dos registradores de 16 *bits* e, por isso, pode parecer estranho. Lembre-se de que `P1DIR` corresponde ao *byte* menos significativo da porta `PADIR`. Por isso, o símbolo `PADIR_L`. O mesmo acontece com `PAOUT`, `PAIN` etc. Outro detalhe, o compilador, por definição prepara os códigos para o endereçamento em 20 *bits* (MSPX). Por isso, surgem as instruções `CALLA` e `RETA`. O Anexo C, no tópico C.10.1, mostra com detalhes como gerar a listagem *assembly* correspondente a um programa em C. Ainda neste apêndice, o tópico C.11 mostra como mesclar programas em C e *assembly*.

Listagem do assembly correspondente ao ER 5.1

```
;*****************************************************************************
;* FUNCTION NAME: io_config                                                  *
;*                                                                           *
;*   Regs Modified     : SP,SR                                               *
;*   Regs Used         : SP,SR                                               *
;*   Local Frame Size  : 0 Args + 0 Auto + 0 Save = 0 byte                   *
;*****************************************************************************
io_config:
;* ---------------------------------------------------------------------------*
        OR.B      #1,&PADIR_L+0         ; [] |20|
        BIC.B     #1,&PAOUT_L+0         ; [] |21|
        OR.B      #128,&PBDIR_H+0       ; [] |23|
        AND.B     #127,&PBOUT_H+0       ; [] |24|
        BIC.B     #2,&PADIR_H+0         ; [] |26|
        OR.B      #2,&PAREN_H+0         ; [] |27|
        OR.B      #2,&PAOUT_H+0         ; [] |28|
        BIC.B     #2,&PADIR_L+0         ; [] |30|
        OR.B      #2,&PAREN_L+0         ; [] |31|
        OR.B      #2,&PAOUT_L+0         ; [] |32|
        RETA                            ; []
                                        ; []
      .sect ".text:main"
      .clink
      .global     main

;*****************************************************************************
;* FUNCTION NAME: main                                                       *
;*                                                                           *
;*   Regs Modified     : SR                                                  *
;*   Regs Used         : SR                                                  *
;*   Local Frame Size  : 0 Args + 0 Auto + 0 Save = 0 byte                   *
;*****************************************************************************
main:
```

```
;* ---------------------------------------------------------------------*
        MOV.W     #23168,&WDTCTL+0      ; [] |8|
        CALLA     #io_config            ; [] |9|
                                        ; [] |9|
        JMP       $C$L2                 ; []
                                        ; []
;* ---------------------------------------------------------------------*
$C$L1:
        OR.B      #128,&PBOUT_H+0       ; [] |13|
;* ---------------------------------------------------------------------*
;*   BEGIN LOOP $C$L2
;* ---------------------------------------------------------------------*
$C$L2:
        BIT.B     #2,&PAIN_H+0          ; [] |11|
        JNE       $C$L3                 ; [] |11|
                                        ; [] |11|
;* ---------------------------------------------------------------------*
        OR.B      #1,&PAOUT_L+0         ; [] |11|
        JMP       $C$L4                 ; [] |11|
                                        ; [] |11|
;* ---------------------------------------------------------------------*
$C$L3:
        BIC.B     #1,&PAOUT_L+0         ; [] |12|
;* ---------------------------------------------------------------------*
$C$L4:
        BIT.B     #2,&PAIN_L+0          ; [] |13|
        JEQ       $C$L1                 ; [] |13|
                                        ; [] |13|
;* ---------------------------------------------------------------------*
        AND.B     #127,&PBOUT_H+0       ; [] |14|
        JMP       $C$L2                 ; [] |14|
                                        ; [] |14|
        NOP                             ; []
;* ---------------------------------------------------------------------*
```

ER 5.2 (ER 3.8). Escreva um programa que implemente um contador binário com os dois *leds* da placa. A cada acionamento de S1 (transição de aberta para fechada), o contador avança uma contagem, de acordo com a referência indicada na tabela abaixo.

Tabela 5.1. Contador de 2 bits construído com LED1 e LED2.

Contagem	*LED*1	*LED*2
0 (00)	Apagado	Apagado
1 (01)	Apagado	Aceso
2 (10)	Aceso	Apagado
3 (11)	Aceso	Aceso

Solução:
Aqui neste exercício precisaremos esperar pelas mudanças de estado da chave S1 e ainda tratar os rebotes. Para esperar o acionamento ou a liberação de S1 usamos as duas linhas abaixo. Note que o `while` é seguido por um ponto-e-vírgula, ou seja, é um `while` com um *statement* vazio, ou seja, nada faz. Quando a chave é acionada, ou liberada, conforme o caso, o laço do `while` termina.

- `while ( (P2IN&BIT1) == BIT1); //Esperar acionar S1`
- `while ( (P2IN&BIT1) == 0);    //Esperar soltar S1`

A listagem abaixo apresenta o programa solução. Ele é simples e dispensa maiores explicações. Vamos apenas comentar a função `debouce()`. Esta função é um contador de zero até o limite que é passado como argumento. Assim, ela apenas gasta tempo. O argumento que é passado dita o quanto de tempo ela gasta. Note o atraso para eliminar o rebote é ditado pela constante `DBC`, definida logo no início do programa.

Agora, vamos a um detalhe importante do compilador C do CCS. Este compilador faz uso de um optimizador que procurar deixar os programas compactos e velozes. Em outras palavras, ele remove códigos ineficazes, remove trechos de códigos sem utilidade. Este é o caso do contador usado na função `debounce()` que apenas gasta tempo. Para evitar esta ação do compilador, declaramos como `volatile` a variável usada. Este modificador `volatile` informa ao compilador para não fazer qualquer tipo de melhoria quando esta variável for usada. O leitor é convidado a rodar o programa removendo este modificador e constatar a presença de rebotes. Fica como sugestão ao leitor examinar a listagem *assembly* que o compilador gera para este programa?

Listagem da solução do ER 5.2

```
// ER 5.2 (ER 3.8)

#include <msp430.h>

#define DBC     1000    //Sugestão para o debounce

// Protótipo das funções
void cont_leds(int ct);
void debounce(int valor);
void io_config(void);

int main(void){
      int cont=0;
      WDTCTL = WDTPW | WDTHOLD;     // stop watchdog timer
      io_config();
      while (1){
          while ( (P2IN&BIT1) == BIT1);   //Acionou S1?
          debounce(DBC);
```

```
          cont++;
          cont_leds(cont);
        while ( (P2IN&BIT1) == 0);      //Soltou S1?
        debounce(DBC);
      }
      return 0;
}

// Acionar os leds
void cont_leds(int ct){
    switch(ct&3){
        case 0: P1OUT &= ~BIT0;     P4OUT &= ~BIT7;     break;  //00
        case 1: P1OUT &= ~BIT0;     P4OUT |=  BIT7;     break;  //01
        case 2: P1OUT |=  BIT0;     P4OUT &= ~BIT7;     break;  //10
        case 3: P1OUT |=  BIT0;     P4OUT |=  BIT7;     break;  //11
    }
}

// Debounce
void debounce(int valor){
    volatile int x;           //volatile evita optimizador do compilador
    for (x=0; x<valor; x++);  //Apenas gasta tempo
}

// Configurar GPIO
void io_config(void){
    P1DIR |= BIT0;      //Led1 = P1.0 = saída
    P1OUT &= ~BIT0;     //Led1 apagado

    P4DIR |= BIT7;      //Led2 = P4.7 = saída
    P4OUT &= ~BIT7;     //Led1 apagado

    P2DIR &= ~BIT1;     //S1 = P2.1 = entrada
    P2REN |= BIT1;      //Habilitar resistor
    P2OUT |= BIT1;      //Habilitar pullup
}
```

ER 5.3 (Repetição do ER 3.11). Escreva um programa que implemente um contador binário com os dois *leds* da placa. A cada acionamento de S1 (transição de aberta para fechada), o contador avança uma contagem e a cada acionamento de S2 (transição de aberta para fechada), o contador recua uma contagem. Em outras palavras:

- S1 → incrementa o contador e
- S2 → decrementa o contador.

Solução:

Agora, não podemos mais ficar esperando por uma chave ser acionada. Precisamos monitorar ambas as chaves e tomar uma ação quando uma delas for acionada. Para saber se uma chave passou de aberta para fechada, é necessário que ela esteja fechada e que o estado anterior fosse (o passado) aberta. Isto significa que vamos precisar de lembrar de qual era o estado anterior da chave, ou seja, qual o estado da chave por ocasião da última consulta.

Uma proposta é a função denominada de `mon_s1()` que monitora a chave S1. Esta função retorna `TRUE` se a chave passou de aberta para fechada e retorna `FALSE` nos demais casos. Isso facilitou em muito o programa principal, que ficou bem curto. Confira o laço principal na solução apresentada adiante. Vamos relembrar que em C, quando a execução de uma função é iniciada, suas variáveis são criadas e quanto a execução termina, suas variáveis são perdidas. As variáveis locais de uma função "não se lembram" do passado. Para o caso do atual problema, precisamos de uma variável que não fosse perdida ao terminar a função. Assim, nessa variável poderíamos armazenar o último estado da chave S1. Isso é conseguido com a declaração de variável estática, mostrada abaixo.

```
static int ps1=ABERTA;
```

Quando a função `mon_s1()` é chamada pela primeira vez, a variável `ps1` é criada e inicializado com a constante `ABERTA`. Depois ela é mantida e preserva seu último valor ao longo das diversas chamadas dessa função. Ela só pode ser acessada de dentro da função onde foi criada. Note então que não é uma variável global, apesar de se parecer bastante com esta.

Uma outra solução para este caso seria fazer uso de uma variável global. Toda variável declarada fora das funções é global e pode ser acessada por qualquer uma das funções que compõem o programa, ou seja, é "vista por todo mundo". Mas as variáveis que guardam o último estado da chave só dizem respeito a uma única função e, por isso, a melhor solução é declará-las como estática. Usar a solução com variável global funciona, mas polui o programa.

É preciso comentar as constantes definidas no início do programa. As duas constantes `TRUE` e `FALSE`, valendo 1 e 0, respectivamente, podem parecer um exagero, mas elas facilitam a legibilidade do programa. Quando são usadas, o programador deixa claro a intenção de uma opção binária ou booleano. Convidamos o leitor ver o caso de variáveis do tipo booleana. Também declaramos duas constantes, `ABERTA` e `FECHADA`, que valem 1 e 0, respectivamente. O leitor pode se perguntar por que usar duas constantes valendo 1 (`TRUE` e `ABERTA`) e duas constantes ue valendo 0 (`FALSE` e `FECHADA`). Isso foi feito para deixar clara intenção de se testar o estado da chave. Ficaria muito estranho falarmos que `FALSE` corresponde a chave aberta.

```
// ER 5.3 (ER 3.11)

#include <msp430.h>

#define TRUE    1
#define FALSE   0
#define ABERTA  1       //Chave aberta
#define FECHADA 0       //Chave fechada
#define DBC     1000    //Sugestão para o debounce

int mon_s1(void);
int mon_s2(void);
void cont_leds(int ct);
void debounce(int valor);
void io_config(void);

int main(void){
    int cont=0;
     WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer
     io_config();
     while (TRUE){
         if (mon_s1() == TRUE)  cont++;   //Chave 1?
         if (mon_s2() == TRUE)  cont--;   //Chave 2?
         cont_leds(cont);
     }
     return 0;
}

// Monitorar S1, retorna TRUE se foi acionada
int mon_s1(void){
    static int ps1=ABERTA;      //Guardar passado de S1
    if ( (P2IN&BIT1) == 0){     //Qual estado atual de S1?
        if (ps1==ABERTA){       //Qual o passado de S1?
            debounce(DBC);
            ps1=FECHADA;
            return TRUE;
        }
    }
    else{
        if (ps1==FECHADA){       //Qual o passado de S1?
            debounce(DBC);
            ps1=ABERTA;
            return FALSE;
        }
    }
    return FALSE;
}

// Monitorar S2, retorna TRUE se foi acionada
```

```
int mon_s2(void){
    static int ps2=ABERTA;      //Guardar passado de S2
    if ( (P1IN&BIT1) == 0){     //Qual estado atual de S2?
        if (ps2==ABERTA){       //Qual o passado de S2?
            debounce(DBC);
            ps2=FECHADA;
            return TRUE;
        }
    }
    else{
        if (ps2==FECHADA){       //Qual o passado de S2?
            debounce(DBC);
            ps2=ABERTA;
            return FALSE;
        }
    }
    return FALSE;
}

// Copiar essas funções dos exercícios anteriores
void cont_leds(int ct){ ...}
void debounce(int valor) { ...}
void io_config(void) { ...}
```

5.4. Uso de Interrupções MSP430 em C

No capítulo anterior estudamos as interrupções e fizemos vários exercícios usando *assembly*. Se o leitor não estiver lembrado, é interessante rever o Capítulo 4 antes de estudar este tópico.

O compilador C oferece uma série de recursos para facilitar nosso trabalho com as interrupções. Elas são denominadas funções intrínsecas e recomendamos uma rápida consulta o item C.5.1 do Apêndice C. Iniciamos apresentando as funções que nos permitem controlar a habilitação geral, ou seja, controlar o estado do bit GIE.

- `__enable_interrupt()` → corresponde à instrução EINT, faz GIE = 1 e
- `__disable_interrupt()` → corresponde à instrução DINT, faz GIE = 0.

Também é possível acessar diretamente o registrador.

- `__bis_SR_register(GIE)` →
- `__bic_SR_register(GIE)` →

Na tabela abaixo é possível ver o *assembly* correspondente e constatar que são inseridos `NOPs` para garantir o correto funcionamento das interrupções.

Tabela 5.2. Alternativas para habilitar e desabilitar interrupções

Função em C	Assembly correspondente

`__enable_interrupt()`	`NOP`
	`EINT`
	`NOP`

`__disable_interrupt()`	`DINT` `NOP`

`__bis_SR_register(GIE)`	`NOP`
	`BIS.W #8,SR`
	`NOP`

`__bic_SR_register(GIE)`	`NOP`
	`BIC.W #8,SR`
	`NOP`

Vamos agora ver como indicamos ao compilador que uma determinada função deve ser usada para atender a uma interrupção, ou seja, como indicar ao compilador para preencher uma posição da tabela de vetores de interrupção com o endereço de uma certa função. Como já afirmamos no capítulo anterior, quem na verdade faz esse preenchimento da tabela é o *linker*. As duas linhas abaixo indicam um exemplo para especificar que a função `minha_isr()` deve atender à interrupção da porta P2.

```
#pragma vector = PORT2_VECTOR
__interrupt void minha_isr(void){ ... }
```

Agora se faz necessário uma série de explicações. Em linguagem C, as diretivas `pragma` são usadas para disponibilizar recursos específicos do compilador ou do sistema operacional. No caso do compilador do CCS foi criada a diretiva `#pragma vector` para indicar ao compilador que o endereço da próxima função deve ser colocado na Tabela de Vetores de Interrupção, na posição indicada (`PORT2_VECTOR`). Já vimos que `PORT2_VECTOR` é uma constante que vale 42, assim, as duas linhas abaixo têm o mesmo efeito.

```
#pragma vector = 42
__interrupt void minha_isr(void){ ... }
```

A palavra `__interrupt` serve para indicar ao compilador que uma determinada função será usada para atender a uma interrupção. Vamos pensar um pouco sobre uma função em C que vai atender a uma interrupção. É esperado que ela não receba argumentos, já que não vai ser chamada da forma convencional. Além disso, ela não pode retornar argumentos, já que não tem quem os receba. Por isso, uma função que atende a uma interrupção sempre vai ter a forma `void funcao(void)`. Tem um outro detalhe importante, que é a preservação do contexto do programa principal. A função de

interrupção não pode alterar os registradores que estão sendo usados pelo programa principal. Por isso, ela os salva na pilha e antes de regressar restaura seus valores. Aí surge uma pergunta: como vou saber quais registradores meu programa em C está usando? É complicado. Na maioria das vezes, a solução é um pouco bruta: salvamos todos os registradores e, antes de regressar da interrupção, restauramos todos eles.

O compilador do CCS faz uma coisa um pouco mais inteligente. Ele salva apenas os registradores usados pela sub-rotina de interrupção. Porém, se esta sub-rotina chamar outras funções, aí a coisa se complica, pois o compilador não temo como saber (de forma simples) quais registradores essas outras funções chamadas usam. Neste caso, ele salva todos os registradores.

A palavra `__interrupt` também orienta o compilador a finalizar a função com um `RETI` e não com o `RET` comum. Se a função que atende à interrupção não terminar com um `RETI`, o processador irá ser perder. Deixamos para o leitor explicar o porquê disso. Sugestão, escreva um programa simples e que use interrupção e omita o modificador `__interrupt`.

ER 5.4 (ER 4.2). Escreva um programa que inverta o estado do Led1 a cada acionamento da chave S1, que é monitorada com o uso de interrupção.

Solução:
Já vimos que não é uma boa ideia monitorar chaves com o uso de interrupções. Porém, para deixar as coisas simples, vamos usar essa solução já que nosso objetivo aqui é ilustrar uso de interrupções em C. É claro que seremos perturbados pelos rebotes.

Como a chave S1 (P2.1) faz um curto para a terra quando acionada, vamos configurar o pino como entrada com *pullup* e habilitar a interrupção por flanco de descida. A listagem abaixo ilustra a solução. O leitor deve prestar atenção à lógica usada para a interrupção. A função que atende à interrupção está marcada em cor abóbora. Ao rodar o programa iremos perceber que ele funciona, mas com os já esperados rebotes na chave S1.

Listagem solução do ER 5.4

```
// ER 5.4
#include <msp430.h>

void io_config(void);

int main(void)
{
      WDTCTL = WDTPW | WDTHOLD;     // stop watchdog timer
      io_config();
      __enable_interrupt();   //Fazer GIE=1
      while(1);               //Laço infinito
    return 0;
```

```
}

#pragma vector=PORT2_VECTOR
__interrupt void isr_p2(void){
    P2IV;           //Apagar pedido (linha 17)
    P1OUT ^= BIT0;  //Inverter Led1 (linha 18)
}

// Configurar GPIO
void io_config(void){
    P1DIR |= BIT0;      //Led1 = P1.0 = saída
    P1OUT &= ~BIT0;     //Led1 apagado

    P4DIR |= BIT7;      //Led2 = P4.7 = saída
    P4OUT &= ~BIT7;     //Led1 apagado

    P2DIR &= ~BIT1;     //S1 = P2.1 = entrada
    P2REN |= BIT1;      //Habilitar resistor
    P2OUT |= BIT1;      //Habilitar pullup
    P2IES |= BIT1;      //Flanco de descida
    P2IFG = 0;          //Apagar possíveis pedidos
    P2IE  |= BIT1;      //Habilitar interrupção
}
```

Precisamos fazer alguns comentários sobre o programa acima. O vetor da porta P2 ocupa a posição 42, como está mostrado na Tabela 4.1 do capítulo anterior. Assim, a declaração `#pragma vector=PORT2_VECTOR` poderia ser substituída por `#pragma vector=42`.

Como nosso programa é muito simples e temos a certeza de que só uma interrupção da porta P2 foi habilitada, não precisamos consultar o registrador P2IV. Entretanto, precisamos apagar o *flag* que provocou a interrupção. Em outras palavras, dentro da função de interrupção precisamos apagar o *bit* 1 do registrador P2IFG. Isto pode ser feito com o *statement* `P2IFG &= ~BIT1`. Entretanto, para deixar mais interessante e ilustrar um conceito fizemos apenas a leitura do registrador P2IV. O leitor deve estar lembrado de que a leitura de P2IV, além de retornar um número que indica a interrupção de maior prioridade pendente, também apaga o *flag* correspondente. É interessante ver que apenas uma linha com o nome do registrador implica na sua leitura.

A listagem abaixo apresenta o *assembly* correspondente apenas à função que atende a interrupção da Porta P2. É interessante notar que a linha 17 do programa em C resultou em `MOV.W  &P2IV+0,r15` que lê o registrador P2IV e o guarda em R15. Como R15 é usado, então uma cópia de seu valor original e guardado na pilha (`PUSH.W  r15`) e ao final da função seu valor original é restaurado (`POP  r15`). Note que a função terminou com um retorno de interrupção: `RETI`.

```
;-------------------------------------------------------------------------------
```

```
;* FUNCTION NAME: isr_p2                                                *
;*                                                                      *
;*   Regs Modified     : SP,SR,r15                                      *
;*   Regs Used         : SP,SR,r15                                      *
;*   Local Frame Size  : 0 Args + 0 Auto + 2 Save = 2 byte              *
;************************************************************************
isr_p2:
;* ---------------------------------------------------------------------*
        PUSH.W    r15                   ; []
        MOV.W     &P2IV+0,r15           ; [] |17|
        XOR.B     #1,&PAOUT_L+0         ; [] |18|
        POP       r15                   ; []
        RETI                            ; []
```

Convidamos agora o leitor a fazer um experimento. Remova o modificador `__interrupt` que está logo após a declaração `#pragma vector`, como mostrado abaixo. O que será que vai acontecer?

```
#pragma vector=PORT2_VECTOR
void isr_p2(void){ ...
```

Com a omissão do modificador `__interrupt` o compilador trata a função `isr_p2()` como uma função ordinária qualquer. Neste caso então, não toma o cuidado de preservar o valor do registrador R15, e pior de tudo, não finaliza a função como um `RETI`, mas com um `RET`. Neste caso (`RET`) o endereço de retorno é retirado do topo da pilha. Mas a chamada da interrupção guardou na pilha primeiro o endereço de regresso e depois o conteúdo de SR. Então o processador vai tomar o último valor de SR como endereço e, é claro, isso não vai dar certo. A listagem abaixo apresenta o *assembly* resultante no caso de omissão do modificador `__interrupt`.

```
************************************************************************
;* FUNCTION NAME: isr_p2                                                *
;*                                                                      *
;*   Regs Modified     : SP,SR,r15                                      *
;*   Regs Used         : SP,SR,r15                                      *
;*   Local Frame Size  : 0 Args + 0 Auto + 0 Save = 0 byte              *
;************************************************************************
isr_p2:
;* ---------------------------------------------------------------------*
        MOV.W     &P2IV+0,r15           ; [] |17|
        XOR.B     #1,&PAOUT_L+0         ; [] |18|
        RET                             ; []
```

Vamos agora fazer uma pequena alteração na rotina que atende a interrupção, de acordo com a listagem abaixo. Note que agora, a função que essa função chama uma outra função apenas para inverter o *led*.

```
...
#pragma vector=PORT2_VECTOR
__interrupt void isr_p2(void){
    P2IV;           //Apagar pedido (linha 18)
    inv_led1();     //Inverter Led1 (linha 19)
}

void inv_led1(void){
    P1OUT ^= BIT0;
}
...
```

Como já vimos, antes de iniciar uma função que atende a uma interrupção, o compilador salva todos os registradores que essa função usa, para que, ao final da função restaurar todos seus valores originais. Entretanto, quando essa função chama uma outra função, o compilador não tem mais como saber quais registradores serão usados e então salva todos. Isto pode ser visto na listagem *assembly* do trecho correspondente à função de interrupção. Foram salvos apenas os registradores de R11 até R15 porque esses são os registradores que o compilador usa para passagem de parâmetros. A preservação do contexto dos registradores de R4 até R10 é de responsabilidade da função chamada. Veja mais detalhes na página 129 do manual slau132o.pdf.

```
;*****************************************************************************
;* FUNCTION NAME: isr_p2                                                     *
;*                                                                           *
;*   Regs Modified     : SP,SR,r11,r12,r13,r14,r15                           *
;*   Regs Used         : SP,SR,r11,r12,r13,r14,r15                           *
;*   Local Frame Size  : 0 Args + 0 Auto + 10 Save = 10 byte                 *
;*****************************************************************************
isr_p2:
;* ---------------------------------------------------------------------------*
        PUSH.W    r15                   ; []
        PUSH.W    r14                   ; []
        PUSH.W    r13                   ; []
        PUSH.W    r12                   ; []
        PUSH.W    r11                   ; []
        MOV.W     &P2IV+0,r15           ; [] |18|
        CALL      #inv_led1             ; [] |19|
        POP       r11                   ; []
        POP       r12                   ; []
        POP       r13                   ; []
        POP       r14                   ; []
        POP       r15                   ; []
        RETI                            ; []
```

Apenas de terminar este tópico, precisamos comentar que, como está ilustrado no pequeno trecho de código abaixo, a função que atende a interrupção também pode ser indicada usando o recurso de `atribute` (página 106 do manual slau132o.pdf).

```
__attribute__((interrupt)) void func( void ).
```

```
...
__attribute__((interrupt(PORT2_VECTOR))) void isr_p2(void){
    P2IV;           //Apagar pedido
    P1OUT ^= BIT0;  //Inverter Led1
}
...
```

ER 5.6 (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 por 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*.

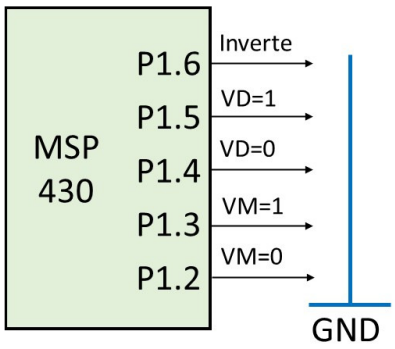

Figura 5.2. Uso de 5 pinos de I/O para controlar os leds por interrupção.

Solução:

No caso deste exercício, temos 5 candidatos para a interrupção da Porta P1. Então, dentro da função atende a essa interrupção é necessário descobrir qual dos 5 candidatos foi o causador da interrupção. Como já vimos, para tais casos o fabricante previu um registrador especial que retorna o número correspondente à interrupção de maior prioridade pendente. Para o caso da Porta P1, este registrador é denominado P1IV e está detalhado na Figura 5.3. Note que a prioridade apenas se aplica quando duas ou mais interrupções da porta P1 estão pendentes.

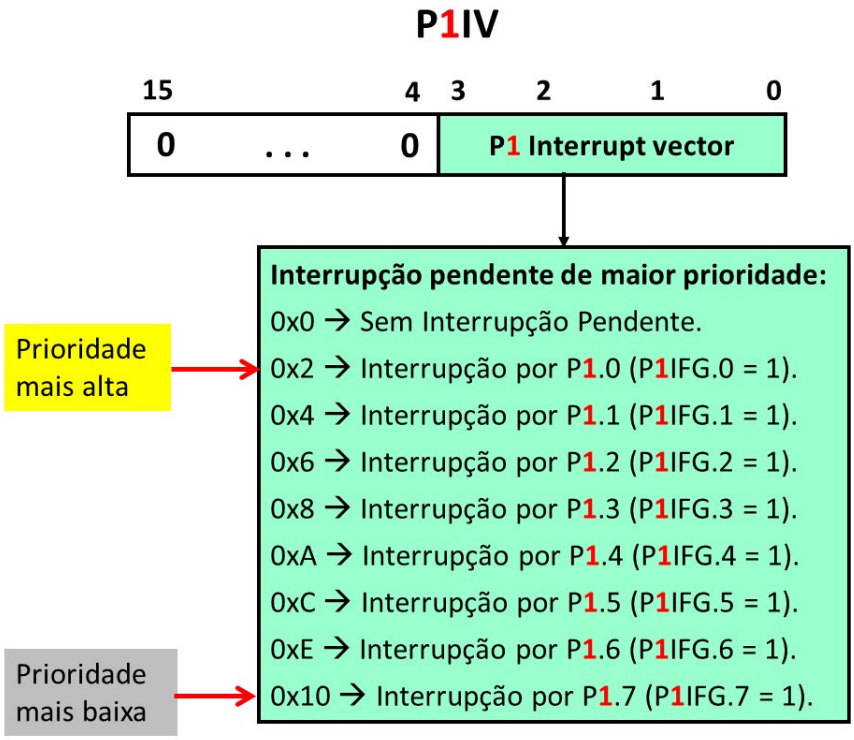

Figura 5.3. Descrição de P1IV: Registrador de Vetor de Interrupção da Porta P1.

A listagem abaixo apresenta o programa solução. Nesta listagem, logo no início, é definida a constante `BIT65432`, para facilitar a configuração da porta P1, já que esses *bits* serão operados. De importante, temos a função que atende à interrupção da porta P1, denominada `isr_p1()`. Esta função faz a leitura do registrador P1IV e depois usa um "`switch and case`" para decidir qual ação tomar. Note que o compilador do CCS prevê uma função intrínseca para a leitura do registrador vetor de interrupção:

```
unsigned int __even_in_range (unsigned int z, unsigned int lim)
```

Esta função retorna o valor de `z`, mas com a restrição de que `z` seja um número par na faixa de 0 até o limite `lim` (página 111 de slau132o.pdf). Ela foi pensada para ser usada com um "`switch and case`". No caso da listagem da solução do ER 5.6, a variável `x` foi

usada para fazer uma ponte, mas isto não é necessário. A linha abaixo indica uma solução mais compacta.

```
    switch(__even_in_range (P1IV,16)){
        case  0:                break;
        case  2:                break;
        ...
    }
```

No caso do exemplo acima, o registrador P1IV é lido e um valor par na faixa de 0 até 16 é retornado para o *statement* `switch`. O Apêndice C, no item C.5.1 apresenta algumas informações interessantes sobre as funções intrínsecas. Apresentamos a seguir a listagem do programa solução.

Listagem da solução do ER 5.6

```
// ER 5.6
#include <msp430.h>

// Constante para facilitar programação
#define BIT65432 (BIT6|BIT5|BIT4|BIT3|BIT2) //0x7C = 0111 1100

void io_config(void);

int main(void){
      WDTCTL = WDTPW | WDTHOLD;    // stop watchdog timer
      io_config();
      __enable_interrupt();        //GIE=1
      while(1);                    //Laço infinito
      return 0;
}

// ISR para a Porta P1
#pragma vector=PORT1_VECTOR
__interrupt void isr_p1(void){
    int x;
    x =__even_in_range (P1IV,16);    //Quem interrompeu?
    switch(x){
        case  0:                break;
        case  2:                break;
        case  4:                break;
        case  6: P1OUT &= ~BIT0; break;  //Apagar  Vermelho
        case  8: P1OUT |=  BIT0; break;  //Acender Vermelho
        case 10: P4OUT &= ~BIT7; break;  //Apagar  Verde
        case 12: P4OUT |=  BIT7; break;  //Acender Verde
        case 14: P1OUT ^=  BIT0; P4OUT ^=  BIT7; break;  //Inverter
        case 16:                break;
    }
```

```
}

// Configurar GPIO
void io_config(void){
    P1DIR |= BIT0;       //Led1 = P1.0 = saída
    P1OUT &= ~BIT0;      //Led1 apagado

    P4DIR |= BIT7;       //Led2 = P4.7 = saída
    P4OUT &= ~BIT7;      //Led1 apagado

    P1DIR &= ~BIT65432;  //P2.65432 = entrada
    P1REN |=  BIT65432;  //Habilitar resistor
    P1OUT |=  BIT65432;  //Habilitar pullup
    P1IES |=  BIT65432;  //Flanco de descida
    P1IFG &= ~BIT65432;  //Apagar possíveis pedidos
    P1IE  |=  BIT65432;  //Habilitar interrupção
}
```

5.5. Exercícios Propostos

O leitor deve ter notado que os Exercícios Resolvidos já foram apresentados. Como Exercícios Propostos, sugerimos que o leitor use a linguagem C para solucionar os Exercícios Propostos nos Capítulos 3 e 4.