19. INTERRUPÇÕES

O controlo simultâneo de vários acontecimentos externos por parte do microprocessador, recorrendo exclusivamente ao teste continuado de entradas, torna o desenho das aplicações complexo, principalmente se tivermos que contabilizar tempos e realizar outros processamentos em simultâneo com a observação de vários eventos externos. Para contornar esta dificuldade, os microprocessadores em geral, põem disponível uma entrada, através da qual se pode accionar um mecanismo denominado por interrupção. Este mecanismo consiste no seguinte: o microprocessador, sempre que termina a execução de uma instrução, testa esse pino de entrada (*Interrup Request* - IRQ) e caso este se encontre activo, gera uma chamada para uma rotina específica. Esta rotina, denominada rotina de serviço – (*Interrupt Sercice Rotine* - ISR), executará uma acção relacionada com o evento que desencadeou o pedido de interrupção, retornando em seguida ao programa que foi interrompido. Do ponto de vista macro, a execução da ISR ocorre assincronamente em relação ao programa principal.

Este mecanismo tem várias implicações na arquitectura do processador, uma vez que é necessário assegurar as seguintes operações:

- Dar à aplicação o controlo de inibição e permissão para o microprocessador aceitar o pedido de interrupção;
- Quando ocorrer a interrupção suspender a execução em curso;
- Preservar o valor do PC corrente para assegurar o retorno ao programa interrompido, bem como do estado de execução do microprocessador, ou seja, preservar o valores de alguns registos do microprocessador;
- Impedir o microprocessador de aceitar nova interrupção.
- Vectorizar o microprocessador para a ISR;

Cada uma destas operações tem diversas formas de implementação.

19.1 Permissão e Inibição

A inibição ou permissão de interrupções consiste na manipulação de uma *flag* que representa o estado de aceitação de interrupções. Esta *flag* é iniciada no estado de inibição.

O microprocessador ao atender uma interrupção tem que ficar automaticamente impedido de aceitar novas interrupções para poder prosseguir executando as instruções da ISR sem ser interrompido descontroladamente. Este facto levanta um problema no retorno ao programa interrompido, uma vez que é necessário repor na íntegra o estado de execução do microprocessador, inclusive o estado de aceitação das interrupções, dai que o retorno e a permissão de interrupção tenham que ser feita de forma indivisível.

19.2 Preservar estado de execução

Quanto ao preservar do valor corrente do PC, é por norma utilizado o mecanismo disponível no microprocessador para dar suporte à implementação de rotinas (STACK ou registo LINK). Quanto à preservação do valor dos restantes registos do microprocessador ou fica ao cuidado do programador, salvaguardá-lo numa estrutura de dados auxiliar, ou existe uma duplicação de registos do microprocessador e que são comutados automaticamente pelo processador ao atender a interrupção.

19.3 Vectorização

Depois de ser salvo o contexto do programa interrompido procede-se à vectorização para a ISR, que consiste em colocar como conteúdo do registo PC o endereço da ISR associada à interrupção.

O estabelecimento do endereço da ISR pode ser feito de várias formas, como por exemplo:

- Valor constante estabelecido pela arquitectura do microprocessador;
- Leitura a partir do barramento de uma instrução (Intel 8085);
- Leitura a partir do barramento de um índice de uma tabela onde são estabelecidos os endereços das respectivas ISRs (Intel x86).

19.4 Retorno de interrupção

No retorno ao programa interrompido a arquitectura tem que providenciar mecanismos que possibilitem simultaneamente o retorno e a reposição do estado de aceitação de interrupções. E novamente aqui existem várias soluções: ou uma instrução de permissão de interrupções com efeito retardado para possibilitar a execução da instrução de retorno com o microprocessador ainda inibido; ou uma instrução específica que execute de forma integrada a reposição do estado do processador e o retorno ao programa interrompido. Também se pressupõe que o processamento realizado pela ISR levou o elemento que gerou a interrupção a retirar o pedido de interrupção.

19.5 Pedido de interrupção

O microprocessador pode disponibilizar uma ou mais entradas para pedido de interrupção (*Interrupt Request* - IRQ). Estas entradas podem ter várias naturezas de excitação, nomeadamente: *level-sensitive*, *edge-trigger-sensitive*, *edge-level-sensitive*. Esta variedade de sensibilidades está directamente associada ao tipo de dispositivos que requerem interrupção. As entradas de interrupção com natureza *level-sensitive* só podem ser utilizadas por periférico que tenham a capacidade de retirar o pedido de interrupção quando forem acedidos pela ISR, caso contrário, deverão utilizar entradas de interrupção com sensibilidade *edge-trigger-sensitive* ou *edge-level-sensitive*, permitindo desta forma que o pedido ainda permaneça no momento de retornar da ISR.

19.6 Interrupções no P16

O P16 põe disponível uma entrada de IRQ denominada nEXINT, activada com valor lógico zero e com natureza de excitação *level-sensitive*, ou seja, requer interrupção enquanto a entrada permanecer activa. Como já foi anteriormente referido, as entradas de interrupção com este tipo de sensibilidade requerem que o periférico não mantenha o pedido no momento de retornar da ISR (ver exemplo 1). O P16 amostra a entrada nEXINT a meio de T3 ou T6 como mostra a Figura 19 -1, e caso esteja activa, transita de T3/T6 para o estado de interrupção TINT, onde desenvolve as acções que lhe estão associadas e que adiante descreveremos. A passagem do microprocessador pelo estado de interrupção é assinalada nos *bits* de estado S1 e S0, possibilitando que o dispositivo que requereu interrupção seja informado antecipadamente da aceitação da interrupção por parte do microprocessador e assim dispor de mais tempo antes de ser acedido. Esta informação pode ser utilizada pelos dispositivos geradores e gestores de interrupções.

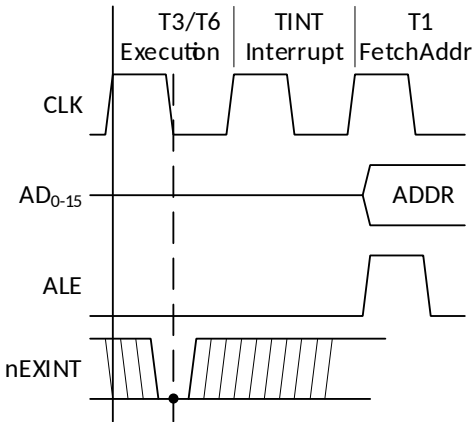


Figura 19-1 Diagrama temporal do estado de interrupção

A introdução do mecanismo de interrupções no P16 implica alterações à estrutura inicial para poder realizar as operações anteriormente descritos. Estas alterações terão uma complexidade acrescida, pelo facto de se pretender que a estrutura responda ao esquema de prioridades *fully nested*. Num esquema de prioridades *fully nested* uma ISR pode ser interrompida por outra interrupção mais prioritária.

As alterações são as seguintes: criação de um segundo registo LR e adição de duas *flags* ao registo CPSR, uma para inibição e permissão de interrupções – denominada I (*Interrupt*) – e outra para indicar se o microprocessador se encontra a servir uma interrupção ou a executar o programa normal – denominada M (*Mode*).

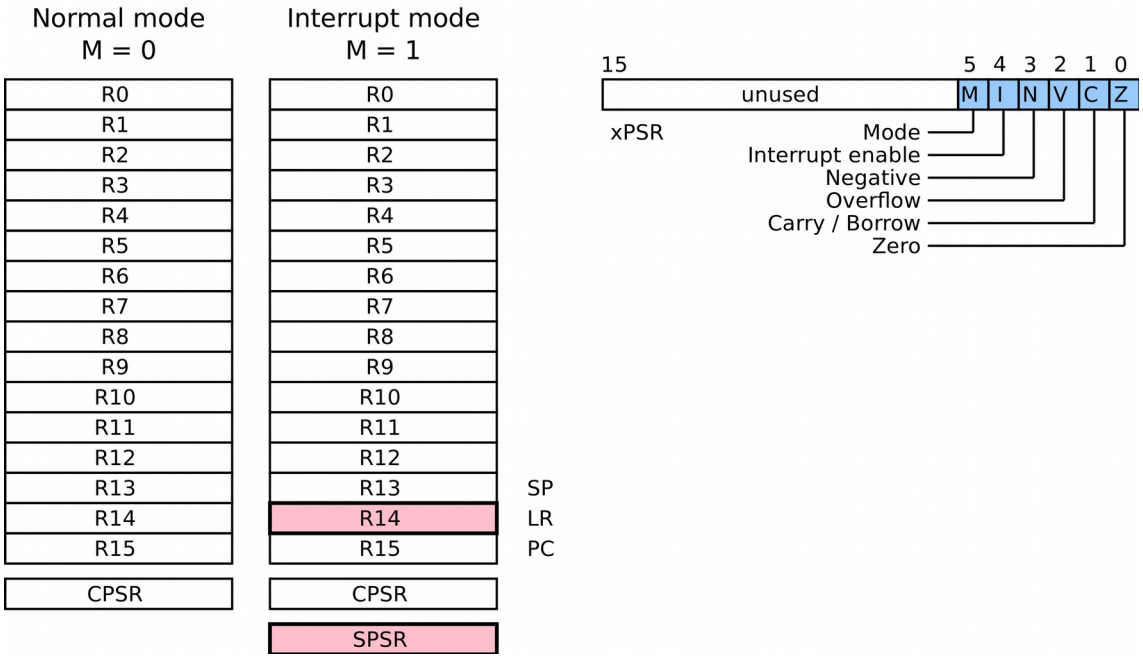


Figura 19- 2: Registos do P16

Aquando de uma interrupção o microprocessador realiza as seguintes acções:

- Copia o valor do registo CPSR para o registo SPSR;

- Coloca a um a *flag* M do registo CPSR mudando o microprocessador do modo normal para modo interrupção. Esta acção comuta o registo LR da aplicação para o registo LR da interrupção;
- Coloca a zero *a flag* I do registo CPSR inibindo assim as interrupções;
- Copia o valor do registo PC para o registo LR do modo de interrupção;
- Coloca o valor dois no registo PC, vectorizando o processamento para o endereço 0x0002, ponto de entrada da interrupção.

Para garantir indivisibilidade entre as várias acções que são necessárias realizar para o retorno da interrupção é utilizada a instrução **movs pc, lr** que, além de copiar para o registo PC o valor do registo LR de modo interrupção, copia o valor do registo SPSR para o registo CPSR. Se o valor do registo SPSR não for alterado pela ISR, esta cópia repõe a permissão de interrupções e o modo normal, que por sua vez implica comutação do registo LR.

19.7 Gestão de interrupções

É natural nos sistemas baseados em microprocessadores existirem várias fontes de interrupção. Como o microprocessador põe disponível uma única entrada para pedido de interrupção, o sistema deverá dispor de um dispositivo periférico que faça a gestão dos vários pedidos de interrupção contemplando: as diferentes sensibilidades, a simultaneidade dos pedidos, a prioridade entre cada um dos pedidos e gerar interrupção ao microprocessador. Este periférico, normalmente denominado por PIC (*Peripheral Interrupt Controller*) põe disponíveis várias entradas de interrupção e permite através de programação:

- Definir a sensibilidade da excitação de cada uma das entradas de interrupção (*Level*, *Edge trigger* ou *Edge Level*);
- Inibir e desinibir individualmente cada entrada de interrupção;
- Definir um qualquer esquema de prioridades (*Linear, Round Robin, Rotation, Fully Nested,* etc.) que assegure que, de entre os vários pedidos de interrupção presentes, é indicado ao microprocessador o índice do mais prioritário, segundo o esquema de prioridades estabelecido.

19.8 Exercícios

Com os exercícios que se seguem pretende-se exemplificar a utilização do mecanismo de interrupções na interacção com dispositivos periféricos. Os exercícios são apresentados numa sequência que visa a introdução gradual dos métodos de programação utilizados em ambiente de interrupções. Em cada exercício começa por se apresentar a solução de programação em linguagem C, para facilitar a descrição e a percepção algorítmica. Em seguida é apresentada a solução de programação em linguagem *assembly*, acompanhada de explicação detalhada dos aspectos relacionados com o processamento das interrupções.

Os esquemas eléctricos assim como o código *assembly* são apresentados com os detalhes que permitem a sua replicação em laboratório sem necessidade de recorrer a outras fontes de informação. Designadamente, todas as ligações são apresentadas em esquema unifilar com indicação das referências dos pinos de ligação, assim como a inclusão de todo o código auxiliar.

Recomenda-se a análise dos exercícios em sequência, porque determinadas operações que se repetem num exercício seguinte não voltam a ser explicadas.

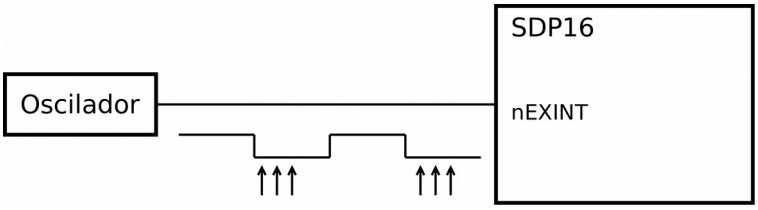
19.8.1 Exercício 1

O relógio de sistema (*system clock*) é um recurso quase sempre disponível num sistema computacional. Esse relógio assume normalmente a forma de um contador que incrementa ou decrementa monotonamente com a passagem do tempo, a um ritmo definido. Os programas de aplicação podem basear-se neste contador para realizar temporizações. O contador pode concretizar-se como um contador *hardware*, uma variável em memória ou ambos.

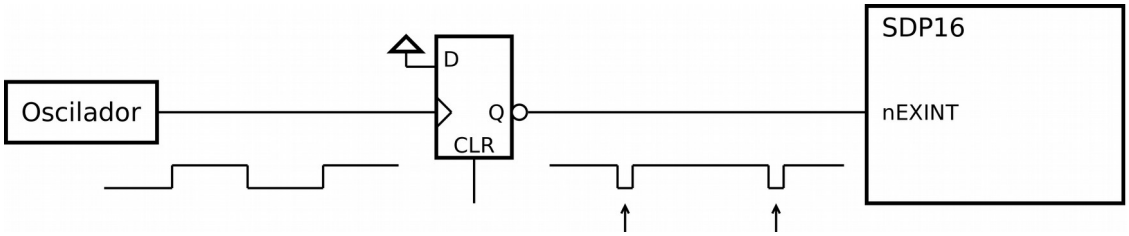
Neste exercício propõe-se a criação de um relógio de sistema baseado numa variável em memória. O incremento dessa variável será realizado numa rotina de atendimento de interrupção – ISR – que será invocada por acção de um sinal de relógio, aplicado à entrada de interrupção do processador.

Para teste do relógio de sistema é utilizado um programa que faz piscar, a um dado ritmo, um LED ligado num *bit* do porto de saída.

Se se aplicasse directamente o sinal de relógio – saída do oscilador – à entrada de interrupção, o pedido de interrupção estaria activo enquanto o sinal estivesse a zero. O que, para um sinal de 100 Hz (valor comummente usado), com *dutty cycle* de 50 %, corresponde a 5 ms. Durante este período a ISR seria executada inúmeras vezes, quando o que se pretende é executá-la apenas uma vez, em cada ciclo do relógio.



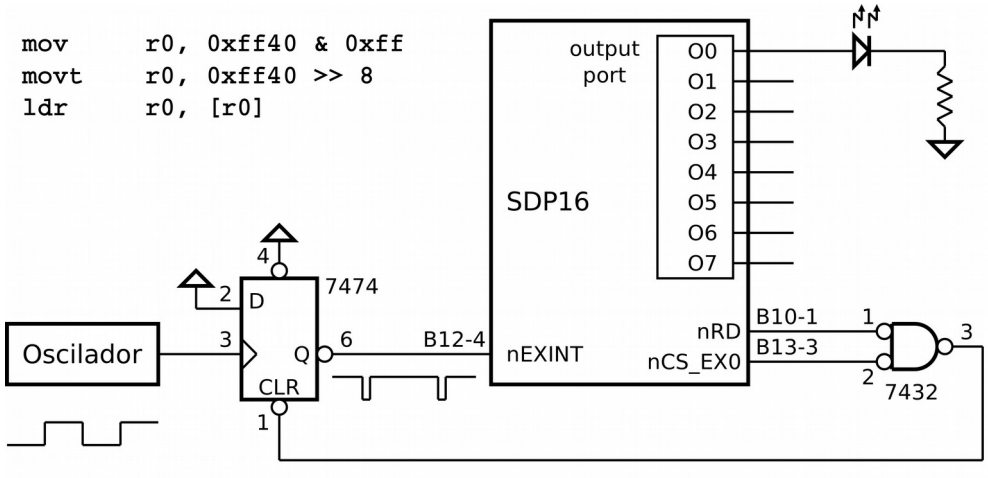
Uma solução possível consiste em intercalar um *flip-flop*, sensível à transição, entre o sinal de relógio e a entrada de interrupção. O que permite controlar o pedido de interrupção, retirando-o após a primeira aceitação – acção CLR sobre o *flip-flop* – e deixar o *flip-flop* receptivo a nova transição e consequentemente a gerar novo pedido de interrupção, apenas num novo ciclo de relógio.



Como realizar a acção CLR sobre o *flip-flop*?

Uma solução consiste em executar na ISR, um acesso a determinado endereço do espaço de entradas/saídas. Em *hardware* a detecção desse acesso provocará a acção CLR.

Por exemplo, na montagem seguinte, para activar o sinal CLR executa-se uma instrução LDR sobre o endereço 0xFF40. (No SDP16, endereçar a 0xFF40 ativa o sinal nCS_EX0.)



No programa seguinte apresenta-se uma solução de programação em linguagem C. O relógio de sistema é concretizado na variável **`system_clock`**. Esta variável é incrementada na ISR em cada invocação desta função – linha 19. No programa principal, é utilizada para realizar um compasso de espera. Na linha 11 é tomado o seu valor na variável **`initial`** e durante o **`while`**, na linha 12, é continuamente calculada a diferença entre o seu valor actual e o valor em **`initial`**. Nas primeiras iterações não haverá diferença, mas à medida que o tempo passa, e as interrupções se vão sucedendo, o valor de **`system_clock`** aumenta e a diferença também aumenta, até atingir o valor **`HALF_PERIOD`**.

A função auxiliar **`port_output`** realiza a escrita no porto de saída.

A função auxiliar **`interrupt_enable`** coloca a *flag* I a um, para permitir ao processador aceitar interrupções.

A função auxiliar **`irequest_clear`** elimina o pedido de interrupção.

```
1     #define     LED_MASK    1
2     #define     HALF_PERIOD 50
3
4     volatile uint16_t system_clock;
5
6     void main() {
7           uint8_t led_state = 0;  /* r4 */
8           interrupt_enable();
9           while (1) {
10                port_output(led_state & LED_MASK);
11                uint16_t initial = system_clock;    /* r5 */
12                while (system_clock - initial < HALF_PERIOD)
13                      ;
14                led_state = ~led_state;
15          }
16    }
17
18    void isr() {
19          system_clock++;
20          irequest_clear();
```

```
21    }
```

PROGRAMA EM ASSEMBLY

Na implementação em linguagem *assembly* poderão observar-se os detalhes de implementação ao nível do processador.

A secção .**startup** contém o código de preparação, adequado à utilização do mecanismo de interrupção. Ao aceitar a interrupção o processador simula a execução de uma instrução **bl** para o endereço **0x0002**. Nesse endereço, deve ser colocado programa que conduza à rotina de atendimento (ISR). No exemplo, esse código corresponde a **ldr pc, add_isr**, na linha 3, que carrega em PC o endereço de **isr**, provocando a passagem para essa rotina.

A função **interrupt_enable** que é traduzida pela sequência[1]

```
mov   r0, IFLAG_MASK
msr   cpsr, r0
```

coloca a *flags* I a um, permitindo ao processador aceitar interrupções. O símbolo **IFLAG_MASK** é equivalente a **0x10** porque a *flag* I ocupa a posição 4 no registo CPSR.

O atendimento de interrupção dá-se depois do processamento de uma qualquer instrução e antes da execução da instrução seguinte, sem que o programador controle o local do programa em que ocorre. Nessa altura todos os registos do processador contêm dados do programa interrompido que não podem ser corrompidos. Os registos LR e CPSR são preservados pelo processador, os restantes ficam a cargo do programador. Antes de utilizar algum registo, a ISR deve salvaguardar o seu conteúdo e restaurá-lo antes de retornar ao programa interrompido. No exemplo, a ISR utiliza apenas os registos R0 e R1, o seu conteúdo prévio é guardado no *stack* à entrada na função pelas instruções PUSH nas linhas 66 e 67 e é reposto à saída pelas instruções POP nas linhas 78 e 79.

O retorno ao programa interrompido é efetivado pela instrução **movs pc, lr** na linha 80. Esta instrução além de colocar o conteúdo de LR em PC e assim retornar ao programa interrompido, restaura também os conteúdos dos registos CPSR e LR. O CPSR por cópia de SPSR e LR por comutação da visibilidade dos registos reais.

A operação de eliminação do pedido de interrupção, representada pela função **irequest_clear**, é realizada nas linhas 75 a 77. Começa-se por carregar em R0 o endereço 0xFF40 e em seguida a execução da instrução **ldr r0,[r0]** ativa simultaneamente os sinais **nCS_EX0** e **nRD**. O local da rotina ISR onde esta operação é realizada é indiferente, porque toda a rotina é executada com o processador em estado de não aceitação de interrupções – *flag* I a zero.

```
1       .section    .startup
```

1 – Na tradução para linguagem *assembly,* o que na linguagem C representa a chamada a uma função, pode não ser traduzido pela chamada a uma rotina *assembly*, pode apenas ser substituido diretamente pelas instruções que realizam a acção da função – tradução *inline* .

```
 2       b     _start
 3       ldr   pc, addr_isr
 4 _start:
 5       ldr   sp, addr_stack_top
 6       ldr   r0, addr_main
 7       mov   r1, pc
 8       add   lr, r1, 4
 9       mov   pc, r0
10       b     .
11 addr_stack_top:
12       .word stack_top
13 addr_main:
14       .word main
15 addr_isr:
16       .word isr
17
18       .text
19
20       .data
21
22       .section    .stack
23       .equ        STACK_SIZE, 64
24       .space      STACK_SIZE
25 stack_top:
26
27 /*-------------------------------------------------------------------------
28 */
29       .data
30 system_clock:
31       .word 0                       ; volatile uint16_t system_clock;
32
33       .equ  LED_MASK, 1
34       .equ  IFLAG_MASK, 0x10
35
36       .equ  HALF_PERIOD, 50
37
38       .text
39 main:
40       mov   r4, 0                   ; uint8_t led_state = 0
41       mov   r0, IFLAG_MASK          ; interrupt_enable();
42       msr   cpsr, r0
43 while:                             ; while (1) {
44       mov   r0, LED_MASK
45       and   r0, r0, r4
46       bl    port_output             ; port_output(led_state & LED_MASK);
47       ldr   r1, addr_system_clock
48       ldrb  r5, [r1]                ; uint16_t initial = system_clock;
49 while1:                            ; while (
50       ldr   r0, [r1]                ;     system_clock - initial
51       sub   r0, r0, r5
52       mov   r2, HALF_PERIOD & 0xff
53       movt  r2, HALF_PERIOD >> 8
54       cmp   r0, r2                  ;      < HALF_PERIOD)
55       blo   while1
56
57
```

```
58        not   r4, r4                  ; led_state = ~led_state;
59
60        b     while
61
62  /*-------------------------------------------------------------------------
63  */
64        .equ  IREQUEST_CLEAR_ADDRESS, 0xff40
65        .text
66  isr:
67        push  r0
68        push  r1
69
70        ldr   r1, addr_system_clock   ; system_clock++;
71        ldr   r0, [r1]
72        add   r0, r0, 1
73        str   r0, [r1]
74
75        mov   r0, IREQUEST_CLEAR_ADDRESS & 0xff   ; irequest_clear();
76        movt  r0, IREQUEST_CLEAR_ADDRESS >> 8
77        ldr   r0, [r0]
78
79        pop   r1
80        pop   r0
81        movs  pc, lr
82
83  addr_system_clock:
64        .word system_clock
85
86  /*-------------------------------------------------------------------------
87  */
88        .equ  SDP16_PORT_ADDRESS, 0xff00
89
90  port_output:
91        mov   r1, SDP16_PORT_ADDRESS & 0xff
92        movt  r1, SDP16_PORT_ADDRESS >> 8
          strb  r0,[r1]
          mov   pc, lr
```

19.8.2 Exercício 2

Baseado no SDP16, e utilizando o mecanismo de interrupções, implementar o controlo de intermitência de um LED. Considere que se dispõe de um botão de pressão ligado num *bit* do porto de entrada e o LED se encontra ligado num *bit* do porto de saída.

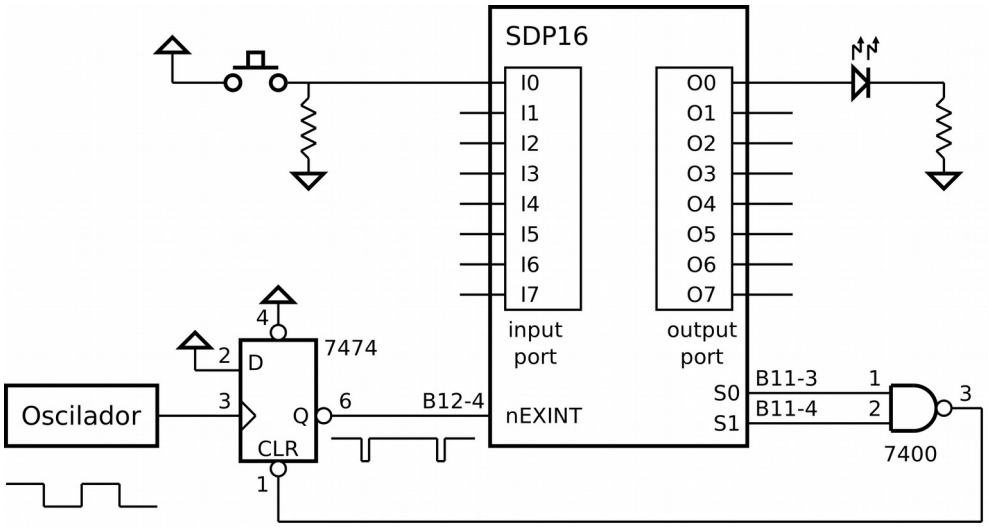
Uma pressão no botão inverte o funcionamento do LED – se estava apagado passa a intermitente e se estava intermitente passa a apagado.

Os estados naturais de um LED são apagado ou aceso. A intermitência é realizada artificialmente por programação, apagando e acendendo o LED a um dado ritmo. Para o programa realizar esta tarefa necessita de uma referência temporal.

A solução proposta consiste em usar um sinal de relógio para gerar interrupção e provocar a execução da ISR a um dado ritmo. Na sua execução, a ISR define o estado do LED. Por exemplo, se se pretender que a intermitência tenha o ritmo de um segundo – meio segundo aceso e meio segundo apagado – o sinal de

relógio deve ter, no mínimo, uma frequência de 2Hz. E assim, a ISR deverá inverter o estado do LED em cada execução.

Uma solução para eliminação do pedido de interrupção consiste em utilizar algum sinal *hardware* que por ventura o processador active a quando da aceitação de interrupção. Este método tem a vantagem de realizar automaticamente a acção CLR do *flip-flop*. No caso do P16, podem-se utilizar os sinais de *status* S0 e S1, que o processador coloca simultaneamente a um, indicando o atendimento de interrupção.



No programa seguinte apresenta-se uma solução de programação em linguagem C. O estado do programa é baseado em duas variáveis: **`blink_state`**, que representa o estado de intermitência do LED – intermitência activa ou intermitência inactiva – e **`led_state`**, que representa o estado instantâneo do LED – apagado ou aceso. No programa principal, entre as linhas 7 e 11, monitoriza-se o estado do botão e age-se sobre a variável **`blink_state`**. Na ISR inverte-se o estado do LED e actualiza-se a sua visualização no porto de saída.

As funções auxiliares **`port_input`** e **`port_output`** realizam respetivamente a leitura do porto de entrada e a escrita no porto de saída. A função auxiliar **`interrupt_enable`** coloca a *flag* I a um para permitir a aceitação de interrupções.

```
1   volatile uint8_t blink_state = 0;
2   uint8_t led_state = 0;
3
3   void main() {
4         port_output(led_state & LED_MASK);
5         interrupt_enable();
6         while (1) {
7               while ((port_input() & BUTTON_MASK) == 0)
8                     ;
9               blink_state = ~blink_state;
10              while ((port_input() & BUTTON_MASK) != 0)
11                    ;
12        }
13  }
14
15  void isr() {
```

```
16          if (blink_state != 0)
17                led_state = ~led_state;
18          else
19                led_state = 0;
20          port_output(led_state  & LED_MASK);
21    }
```

PROGRAMA EM ASSEMBLY

Na implementação em linguagem *assembly* poderão observar-se os detalhes de implementação ao nível do processador.

As funções **port_input** e **port_output** são traduzidas pelas sequência de instruções

```
      ldr  r1, addr_port
      ldrb r0, [r1]
```

e

```
      ldr  r1, addr_port
      strb r0, [r1]
```

respectivamente.

Na ISR apenas são utilizados os registos R0 e R1 por isso apenas estes são salvos no *stack* – linhas 70, 71, 90 e 91.

O pedido de interrupção é eliminado automaticamente por *hardware. D*aí a ausência de programação para esse fim na ISR.

```
1           .section .startup
2           b     _start
3           ldr   pc, addr_isr
4     _start:
5           ldr   sp, addr_stack_top
6           ldr   r0, addr_main
7           mov   r1, pc
8           add   lr, r1, 4
9           mov   pc, r0
10          b     .
11    addr_stack_top:
12          .word stack_top
13    addr_main:
14          .word main
15    addr_isr:
16          .word isr
17
18          .text
19
20          .data
21
22          .section .stack
23          .equ  STACK_SIZE, 64
24          .space      STACK_SIZE
25    stack_top:
26
```

```
27  /*--------------------------------------------------------------------------
28  */
29        .data
30  blink_state:
31        .byte 0     ; uint8_t blink_state;
32  led_state:
33        .byte 0     ; uint8_t led_state;
34
35        .equ  BUTTON_MASK, 1
36        .equ  LED_MASK, 1
37        .equ  IFLAG_MASK, 0x10
38
39        .equ  SDP16_PORT_ADDRESS, 0xff00
40
41        .text
42  main:
43        ldr   r1, addr_led_state      ; port_output(led_state & LED_MASK);
44        ldrb  r0, [r1]
45        mov   r1, LED_MASK
46        and   r0, r0, r1
47        ldr   r1, addr_port
48        strb  r0, [r1]
49        mov   r0, IFLAG_MASK          ; interrupt_enable();
50        msr   cpsr, r0
51        mov   r2, BUTTON_MASK
52  while:                              ; while (1) {
53  while1:                             ; while ((
54        ldr   r1, addr_port           ;     port_input()
55        ldrb  r0, [r1]
56        and   r0, r0, r2              ;      & BUTTON_MASK)
57        bzs   while1                  ;           == 0)
58        ldr   r1, addr_blink_state
59        ldrb  r0, [r1]                ; blink_state = ~blink_state;
60        not   r0, r0
61        strb  r0, [r1]
62  while2:                             ; while ((
63        ldr   r1, addr_port           ;     port_input()
64        ldrb  r0, [r1]
65        and   r0, r0, r2              ;      & BUTTON_MASK)
66        bzc   while2                  ;           != 0)
67        b     while
68
69  isr:
70        push  r0
71        push  r1
72
73        ldr   r1, addr_blink_state    ; if (blink_state != 0)
74        ldrb  r0, [r1]
75        add   r0, r0, 0
76        ldr   r1, addr_led_state
77        beq   isr_if_else
78        ldrb  r0, [r1]                ; led_state = ~led_state;
79        not   r0, r0
80        b     isr_if_end
81  isr_if_else:
82        mov   r0, 0                   ; led_state = 0;
```

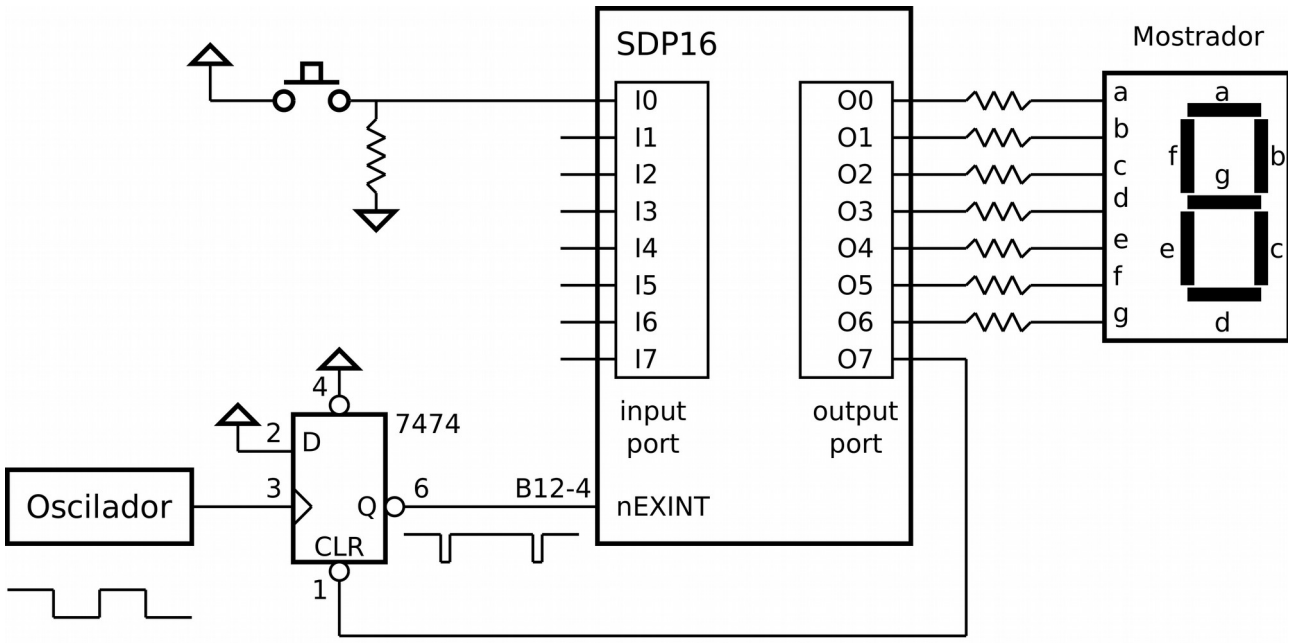
```
83    isr_if_end:
84          strb  r0, [r1]
85          mov   r1, LED_MASK            ; port_output(led_state  & LED_MASK);
86          and   r0, r0, r1
87          ldr   r1, addr_port
88          strb  r0, [r1]
89
90          pop   r1
91          pop   r0
92          movs  pc, lr
93
94    addr_blink_state:
95          .word blink_state
96
97    addr_led_state:
98          .word led_state
99
100   addr_port:
          .word SDP16_PORT_ADDRESS
```

19.8.3 Exercício 3

Tendo como base o SDP16, propõe-se a realização de um cronómetro de segundos, em contagem descendente. A contagem começa com um valor pré-definido, termina em zero e é iniciada ou reiniciada sempre que o botão é premido.

O valor corrente do cronómetro é visualizado num mostrador formado por um dígito de sete segmentos, ligado nos *bits* O0 a O6 do porto de saída. O estado do botão é recolhido no *bit I0* do porto de entrada.

Como solução para eliminação do pedido de interrupção – *clear* do *flip-flop* – é utilizado o *bit* O7 do porto de saída. Esta solução tem a vantagem de não necessitar de *hardware* adicional e a desvantagem de o programa ter que realizar operações de contextos diferentes sobre o mesmo porto de saída.



No programa principal começa-se por afixar o valor zero no mostrador, ao que se segue a detecção de pressão do botão. Admitindo que a pressão do botão provoca a ida do sinal de entrada para o valor lógico um, esse evento provoca a transição do programa da linha 9 para a linha 15. Neste trânsito, são realizadas as acções necessárias para desencadear uma contagem, designadamente:

- linha 11 – iniciar o contador de tempo (variável **time**);
- linha 12 – afixar o valor inicial no mostrador;
- linha 13 – eliminar eventual pedido de interrupção devido a transição de relógio anterior;
- linha 14 – permitir a aceitação das interrupções que farão evoluir a contagem.

Admitindo que a frequência do relógio aplicado à entrada de interrupção é de 1 Hz, a ISR será executada de um em um segundo.

No início do processamento da ISR, começa-se por decrementar o contador de tempo e verificar se atingiu o valor final – zero. Em caso afirmativo, inibe-se a aceitação de novas interrupções através da função **interrupt_clear**. Até haver nova pressão no botão e ser executada a acção **interrupt_enable** não haverá mais interrupções e o valor zero permanecerá afixado no mostrador,

A função **display_write**, invocada na linha 23, atualiza o mostrador com o valor atual do contador de tempo.

A função **irequest_clear**, invocada nas linhas 13 e 24, elimina o pedido de interrupção presente no *flip-flop*, pulsando a zero o *bit* do porto de saída ligado à entrada CLR do *flip-flop*.

```
1    #define     TIME_MAX    9
2    #define     BUTTON_MASK 1
3
4    volatile uint8_t time;
5
6    void main() {
7          display_write(0);
8          while (1) {
9                while ((port_input() & BUTTON_MASK) == 0)
10                     ;
11               time = TIME_MAX;
12               display_write(time);
13               irequest_clear();
14               interrupt_enable();
15               while ((port_input() & BUTTON_MASK) != 0)
16                     ;
17         }
18   }
19
20   void isr() {
21         if (--time == 0)
22               interrupt_disable();
23         display_write(time);
24         irequest_clear();
25   }
26
27   uint8_t port_image;
```

```
28
29  #define DISPLAY_MASK    0x7f
30
31  const uint8_t bin7seg[] =
32        {0x3f, 0x06, 0x5b, 0x4f, 0x66, 0x6d, 0x7d, 0x07, 0x7f, 0x6f};
33
34  void display_write(uint8_t value) {
35        port_image &= ~DISPLAY_MASK;
36        port_image |= bin7seg[value];
37        port_output(port_image);
38  }
39
40  #define IREQUEST_CLEAR_MASK   0x80
41
42  void irequest_clear() {
43        port_output(port_image & ~IREQUEST _CLEAR_MASK);
44        port_output(port_image |= IREQUEST_CLEAR_MASK);
45  }
```

Como o acesso ao porto de saída implica a escrita simultânea dos 8 *bits* – unidade mínima de endereçamento no P16 – coloca-se o problema de atualizar o mostrador sem fazer *clear* ao *flip-flop* ou o de fazer *clear* ao *flip-flop* sem perturbar a imagem no mostrador.

Numa situação simples como a deste exemplo, em que ambas as operações são realizadas no contexto da ISR, o problema resolve-se facilmente reunindo os dados a escrever no mostrador com o comando do *flip-flop*.

Com o objetivo de executar estas operações em contextos independentes, a programação das funções **`display_write`** e **`irequest_clear`** deve integrar a solução deste problema.

Para que num dado contexto, se possa manter o estado do porto de saída, relativo a outros contextos, deve-se conhecer esse estado. Por exemplo, para que ao fazer *clear* ao *flip-flop,* não se altere o mostrador é necessário conhecer o que este tem afixado para tornar a escrever o mesmo valor.

Como não é possível ler de um porto de saída – a não ser que este esteja ligado a um porto de entrada – na implementação destas funções recorre-se à utilização da variável **`port_image`** para guardar o valor atual do porto de saída. Cada função modifica apenas o seu conjunto de *bits* e mantém os restantes. Sendo a atualização do porto de saída, acompanhada da atualização desta variável.

Programa em Assembly

As funções **`port_input`**, **`port_output`** e **`interrupt_enable`** têm implementação igual às dos exercícios anteriores.

A função **`interrupt_disable`** é traduzida pelas instruções

```
mrs   r0, spsr
mov   r1, ~IFLAG_MASK
and   r0, r0, r1
msr   spsr, r0
```

Como é executada no contexto da ISR, atua sobre o registo SPSR, que é a cópia do CPSR do programa interrompido. Ao terminar a ISR, a instrução **`movs pc, lr`** copia o conteúdo de SPSR para CPSR colocando a *flag* I a zero e assim inibindo a aceitação de novas interrupções. Na eventualidade de ser necessário implementar esta função no contexto do programa principal esta deverá atuar diretamente sobre o registo CPSR.

Os conteúdos de R0, R1, R2, R3 e LR são salvaguardados no início da ISR, nas linhas 40 a 44, porque no corpo da ISR irão ser invocadas as funções **`display_wite`** e **`irequest_clear`**. Segundo o protocolo de chamada a funções, os registos R0 a R3 são utilizados para passagem de argumentos e podem ser alterados pela função chamada. Os restantes registos são salvaguardados pela função chamada, caso venham a ser utilizados, dispensando a sua salvaguarda na ISR. O registo LR é modificado pela própria instrução BL, por isso também tem que ser salvaguardado na ISR.

Se a própria função ISR utilizar algum registo de ordem superior a R3, terá ela própria que o salvaguardar.

```
1           .equ  TIME_MAX, 9
2
3           .equ  BUTTON_MASK, 1
4           .equ  IFLAG_MASK, 0x10
5           .equ  PORT_ADDRESS, 0xff00
6
7           .data
8     time:
9           .byte 0     ; uint_t time = 0;
10
11          .text
12    main:
13          mov   r0, 0
14          bl    display_write
15    while:                              ; while (1) {
      while1:                             ; while ((
16          ldr   r1, addr_port           ;     port_input()
17          ldrb  r0, [r1]
18          mov   r2, BUTTON_MASK
19          and   r0, r0, r2        ;      & BUTTON_MASK)
20          bzs   while1                  ;           == 0)
21          mov   r0, TIME_MAX            ; time = TIME_MAX;
22          ldr   r1, addr_time
23          strb  r0, [r1]
24          bl    display_write
25          bl    irequest_clear
26          mov   r0, IFLAG_MASK          ; interrupt_enable();
27          msr   cpsr, r0
28    while2:                             ; while ((
29          ldr   r1, addr_port           ;     port_input()
30          ldrb  r0, [r1]
31          mov   r2, BUTTON_MASK
32          and   r0, r0, r2              ;      & BUTTON_MASK)
33          bzc   while2                  ;           != 0)
34          b     while
35
36    /*-------------------------------------------------------------------------
```

```
37  */
38        .text
39  isr:
40        push  r0
41        push  r1
42        push  r2
43        push  r3
44        push  lr
45
46        ldr   r1, addr_time           ; if (--time == 0)
47        ldrb  r0, [r1]
48        sub   r0, r0, 1
49        strb  r0, [r1]
50        bzc   isr_if_end
51        mrs   r2, spsr                ; interrupt_disable();
52        mov   r1, ~IFLAG_MASK
53        and   r2, r2, r1
54        msr   spsr, r2
55  isr_if_end:
56        bl    display_write           ; display_write(time);
57
58        bl    irequest_clear
59        pop   lr
60        pop   r3
61        pop   r2
62        pop   r1
63        pop   r0
64        movs  pc, lr
65
66  addr_time:
67        .word time
68
69  /*-------------------------------------------------------------------------
70  */
71        .data
72  port_image:
73        .byte 0
74
75        .equ  DISPLAY_MASK, 0x7f
76
77        .text
78  display_write:
79        ldr   r1, addr_port_image     ; port_image &= ~DISPLAY_MASK;
80        ldrb  r2, [r1]
81        mov   r3, ~DISPLAY_MASK
82        and   r2, r2, r3
83        ldr   r3, addr_bin7seg        ; port_image |= bin7seg[value];
84        ldrb  r0, [r3, r0]
85        or    r2, r2, r0
86        strb  r2, [r1]
87        ldr   r1, addr_port           ; port_output(port_image);
88        strb  r2, [r1]
89        mov   pc, lr
90
91  bin7seg:
92        .byte 0x3f, 0x06, 0x5b, 0x4f, 0x66, 0x6d, 0x7d, 0x07, 0x7f, 0x6f
```

```
93
94  addr_bin7seg:
95        .word       bin7seg
96
97  /*-------------------------------------------------------------------------
98  */
99        .equ  IREQUEST_CLEAR_MASK, 0x80
100       .text
101 irequest_clear:
102       ldr   r1, addr_port_image
103       ldrb  r0, [r1]
104       mov   r3, ~IREQUEST_CLEAR_MASK
105       and   r3, r0, r3
106       ldr   r2, addr_port
107       strb  r3, [r2]                ; port_output(port_image & ~CLEAR_MASK);
108       mov   r3, IREQUEST_CLEAR_MASK
109       or    r3, r0, r3
110       strb  r3, [r2]
111       strb  r3, [r1]                ; port_output(port_image |= CLEAR_MASK);
112       mov   pc, lr
113
114 addr_port:
115       .word PORT_ADDRESS
116
117 addr_port_image:
118       .word port_image
```