Índice

Introdução

Um periférico de microprocessador pode ser encarado como um processo que evolui em paralelo com o programa em execução.

Em relação aos periféricos o programa:

- controla o comportamento;
- averigua o estado;
- transfere dados de e para a memória principal.

O programa toma conhecimento do estado do periférico:

- inquirindo o periférico
- ou é avisado quando há mudança de estado relevante – interrupção.

A transferência de dados entre os periféricos e a memória principal pode ser feita: pelo processador, lendo dum lado e escrevendo no outro, usando os registos internos para depósito intermédio; por DMA (Direct Memory Access), sem a intervenção do processador, ser feita directamente entre a memória e o periférico.

O mecanismo de interrupções baseia-se num sinal do periférico para o processador. Este sinal é activado pelo periférico numa mudança de estado e é sentido pelo processador. Ao reconhecer o sinal de interrupção, o processador salta para um endereço pré-estabelecido, alterando o curso normal de processamento.

O código que é executado como consequência de uma interrupção designa-se rotina de atendimento de interrupção (*interrupt service rotine* – ISR).

O processador têm uma entrada principal de interrupção (é comum existirem mais algumas entradas com propósitos especiais). No caso de existirem vários periféricos, cada um gera um pedido de interrupção. Para reunir os vários pedidos e gerar um único pedido ao processador, é utilizado um circuito intermédio – o controlador de interrupções.

Como num micro-sistema existem vários periféricos, põe-se o problema de, face a uma interrupção, seleccionar o código de atendimento adequado ao periférico. Essa selecção pode ser feita por *software*, baseada na pesquisa de estado dos periféricos, ou pelo controlador de interrupções. A utilização de um ou outro critério influencia o tempo de resposta à interrupção.

Os gestores de periféricos são módulos de *software* que, por um lado proporcionam uma interface cómoda para os programas de utilização do periférico, e, por outro funcionam como extensões ao próprio periférico. A interrupção é o mecanismo que desencadeia este processamento de extensão ao periférico, que se processa assincronamente em relação à aplicação.

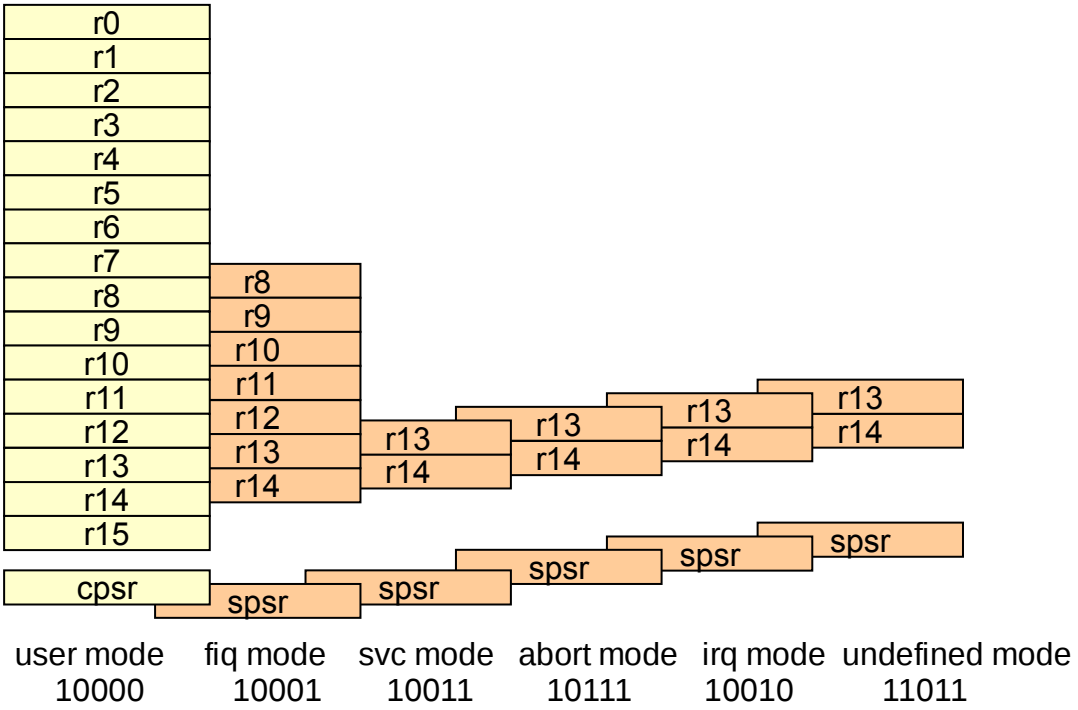
Interrupções na arquitectura ARM

O processador ARM possui duas entradas de interrupção – IRQ e FIQ. Como resposta à activação dum destes sinais, e se o atendimento de interrupções estiver desinibido (*flags* F e I a 0), o processador transfere a execução para os endereços 0x00000018 ou 0x0000001C, respectivamente, simulando a execução de uma instrução bl. Nestas posições de memória estão instruções que carregam no PC o endereço da ISR.

```
    ldr         pc, reset_handler_address
    ldr         pc, undefined_handler_address
    ldr         pc, swi_handler_address
    ldr         pc, prefetch_abort_handler_address
    ldr         pc, data_abort_handler_address
    .word 0
    ldr         pc, irq_handler_address
    ldr         pc, fiq_handler_address

    reset_handler_address:              .word reset_handler
    undefined_handler_address:          .word undefined_handler
    swi_handler_address:                .word swi_handler
    prefetch_abort_handler_address:     .word prefetch_abort_handler
    data_abort_handler_address:         .word data_abort_handler
    irq_handler_address:                .word irq_handler
    fiq_isr_address:                    .word fiq_isr
```

Ao atender uma interrupção o processador comuta para o modo IRQ ou para o modo FIQ.
A mudança de modo implica a utilização de um banco de registos modificado. No modo IRQ os registos LR e SP são substituídos por registos próprios deste modo e no modo FIQ há a acrescentar a substituição dos registos R8 a R12.



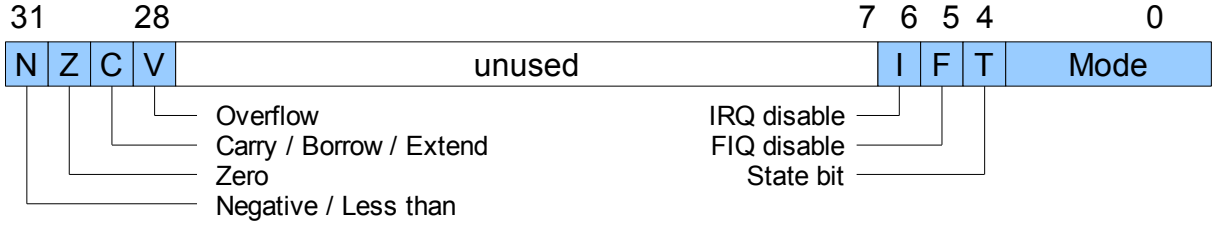
Em cada modo, excepto no de utilizador, existe também um registo para salvaguarda do CPSR do modo anterior – SPSR (Saved Processor State Register). Assim, para regressar ao modo e estado anterior basta repor o SPSR em CPSR.

```
mrs   r0, spsr   Move o conteúdo de SPSR para o registo r0
msr   cpsr, r0   Move o conteúdo de r0 para CPSR
```

A substituição de LR é um imperativo de funcionamento, pois este registo pode estar a ser usado pelo programa interrompido. O endereço de retorno, que LRirq apresenta, é o valor do PC na altura da interrupção – duas instruções à frente da última instrução executada (transparece aqui o funcionamento em *pipeline* do processador). O endereço para onde o processador deve retomar o processamento, após o processamento da ISR, à obtido subtraindo 4 ao valor de LR.

```
irq_handler:
      . . .
      subs      pc, lr, #4
```

A notação '**s**' na instrução **subs** significa que o registo **SPSR** do modo IRQ irá ser transferido para **CPSR** depois do registo PC ter sido actualizado. O **CPSR**, ao receber o valor que possuía antes da interrupção, coloca o processador no mesmo modo e com as *flags* de interrupção no estado original.



No atendimento de interrupções é usual utilizar-se um *stack* separado para o processamento das interrupções de modo a minimizar o consumo de *stack* do programa interrompido. A existência de um registo SP próprio para os modos de interrupção permite a comutação automática de *stack* sem custo de processamento e com um consumo nulo no *stack* do programa interrompido.

Ao atender uma interrupção (IRQ ou FIQ) o processador colocas as flags I ou F a 1, inibindo o respectivo atendimento, e impedindo o atendimento sucessivo.

Para programar o processamento das interrupções em linguagem C há duas soluções: usar atributos do compilador para gerar código específico na codificação das rotinas de interrupção:

```
__attribute__((interrupt("IRQ"))) void isr_handler() {
      ...
}
```

ou usar uma rotina intermédia programada em Assembly:

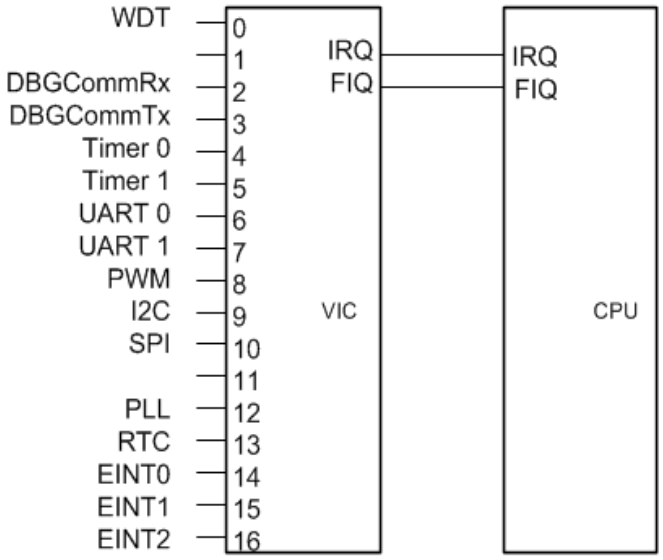
```
      .global isr_handler
isr_handler:
      sub         lr, lr, #4
      stmdb sp!, {r0 - r3, ip, lr}

      bl          isr

      ldmia sp!, {r0 - r3, ip, pc}^
```

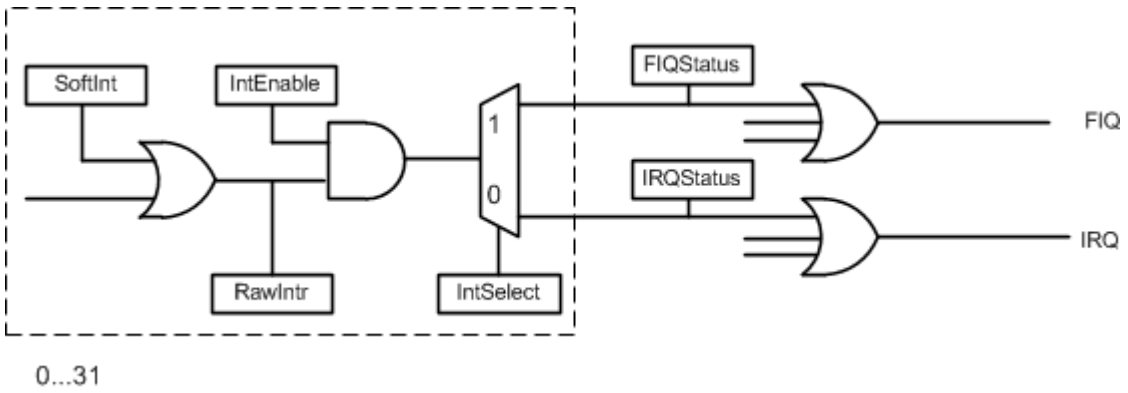
Controlador de interrupções

O controlador de interrupções recebe os pedidos de interrupção vindos dos periféricos e encaminha-os para o processador através dos sinais IRQ ou FIQ.
Na família LPC2XXX o controlador de interrupções é designado por VIC (Vectored Interrupt Controller).



Funcionamento em modo não vectorizado

O controlador de interrupções funciona em modo não vectorizado se os registos **VICVectCntl0..15** permanecerem com o valor zero – estado após *reset*.



Os pedidos de interrupção podem ser gerados artificialmente através dos registos **VICSoftInt** e **VICSoftIntClear**.

O registo **VICRawIntr** permite consultar o estado das fontes de interrupção. É usado para teste, pois permite averiguar se o pedido de interrupção, vindo do periférico, chega ao controlador de interrupções.

Os pedidos de interrupção podem ser permitidos ou inibidos através dos registos **VICIntEnable** e **VICIntEnClear**, respectivamente.

Através do registo **VicIntSelect**, cada fonte de interrupção pode ser configurada para activar FIQ ou IRQ.

A rotina de serviço a IRQ determina quais os periféricos que activaram o pedido através do registo **VICIRQStatus**.

No caso, pouco provável, de se associar mais do que uma fonte de interrupção à entrada FIQ, a rotina de serviço deve consultar **VICFIQStatus** para determinar qual foi o periférico que gerou o pedido.

Preparação

Como preparação para o atendimento de interrupções não vectorizadas é necessário:

- Formar a tabela de excepções

```
    ldr         pc, reset_handler_address
    ldr         pc, undefined_handler_address
    ldr         pc, swi_handler_address
    ldr         pc, prefetch_abort_handler_address
    ldr         pc, data_abort_handler_address
    .word 0
    ldr         pc, irq_handler_address
    ldr         pc, fiq_handler_address

    reset_handler_address:              .word reset_handler
    undefined_handler_address:          .word undefined_handler
```

```
swi_handler_address:                .word swi_handler
prefetch_abort_handler_address:     .word prefetch_abort_handler
data_abort_handler_address:         .word data_abort_handler
                                    .word 0
irq_handler_address:                .word irq_handler
fiq_handler_address:                .word fiq_handler
```

- Coloca a tabela de excepções visível no endereço 0x00000000;

```
io_write_u32(LPC210X_SCB + LPC2XXX_MEMMAP, 2);
```

- Associar a fonte de interrupção à entrada IRQ ou FIQ do processador;

```
io_write_u32(LPC210X_VIC + LPC2XXX_VICIntSelect, 0);
```

- Desinibir o atendimento de interrupções IRQ no processador;

```
mrs   r0, cpsr
and   r0, r0, #~(1 << 7)
msr   cpsr_c, r0
```

- Desinibir o atendimento da fonte de interrupção;

```
io_write_u32(LPC210X_VIC + LPC2XXX_VICIntEnable, 1 << VIC_SOURCE_...);
```

- Iniciar o registo SP do modo IRQ.

```
ldr   r0, =__stack_end__

mov   r1, #0xd2         /* modo IRQ */
msr   CPSR_c, r1
mov   sp, r0
sub   r0, r0, #IRQ_STACK_SIZE

...

mov   r1, #0xdf         /* modo system; disable interrupts */
msr   CPSR_c, r1
mov   sp, r0

...
```

Ciclo de atendimento de interrupção

Um ciclo de atendimento de interrupção tem início no pedido, por parte do periférico. Se a respectiva entrada não estiver inibida irá provocar a activação de IRQ. Como resposta o processador transfere o processamento para **0x00000018**, neste local encontra-se uma instrução que carrega no PC o endereço da rotina de atendimento. O processador passa para o modo IRQ com a *flag* I a 1 - interrupções inibidas. A rotina de atendimento entra em funcionamento e determina o periférico que originou o pedido. Nesta pesquisa está implícito o emprego de um critério de prioridade.

No final do atendimento do periférico, a fonte de interrupção está desactivada (não é obrigatório que assim seja, mas é o mais provável). Como o atendimento decorreu com a *flag* I a 1, o processador não reagiu a IRQ. Ao retornar ao programa interrompido, é restabelecida a *flag* I a 0 e o processador fica apto a iniciar novo ciclo de atendimento.

Exemplo

Incrementar o valor de um contador, por via de uma interrupção, originada no *timer* 0, ao ritmo de uma interrupção por segundo.

```
#include "lpc2106.h"
#include "io.h"

void isr_asm();

/*-------------------------------------------------------------------------
      Quando ocorre match gera interrupão e faz reset a TC
      O periodo é especificado em milisegundos
*/
static void timer_init(U32 period) {
      io_write_u32(LPC210X_T0 + LPC2XXX_TxTCR, LPC2XXX_TxTCR_CTR_RESET);
      io_write_u32(LPC210X_T0 + LPC2XXX_TxPR, PCLK / 1000);
      io_write_u32(LPC210X_T0 + LPC2XXX_TxMCR,
            LPC2XXX_TxMCR_MR1_RESET | LPC2XXX_TxMCR_MR1_INT);
      io_write_u32(LPC210X_T0 + LPC2XXX_TxMR1, period);
      io_write_u32(LPC210X_T0 + LPC2XXX_TxTCR, LPC2XXX_TxTCR_CTR_ENABLE);
}

/*-------------------------------------------------------------------------
      Iniciar a infraestrutura de atendimento de interrupções
*/
static void interrupt_init() {
      /* Inibir todas as interrupções */
      io_write_u32(LPC210X_VIC + LPC2XXX_VICIntEnClear, 0xffffffff);

      /* Limpar eventiuais pedidos de interrupção via software */
      io_write_u32(LPC210X_VIC + LPC2XXX_VICSoftIntClear, 0xffffffff);

      /* Mapear a tabela de interrupções, residente no inicio da RAM,
            no endereço 0x0000000 */
      io_write_u32(LPC210X_SCB + LPC2XXX_MEMMAP, 2);

      /* Associar todas as fontes de interrupção à entrada IRQ */
      io_write_u32(LPC210X_VIC + LPC2XXX_VICIntSelect, 0);

      /* Permitir que hajam interrupções ao nível do processador */
      interrupt_enable();
}

/*-------------------------------------------------------------------------
      Mascarar um pedido de interrupção ao controlador de interrupções
*/
void interrupt_mask(int irq) {
      io_write_u32(LPC210X_VIC + LPC2XXX_VICIntEnClear, 1 << irq);
}

/*-------------------------------------------------------------------------
      Desmascarar um pedido de interrupção ao controlador de interrupções
*/
void interrupt_unmask(int irq) {
      io_write_u32(LPC210X_VIC + LPC2XXX_VICIntEnable, 1 << irq);
}
```

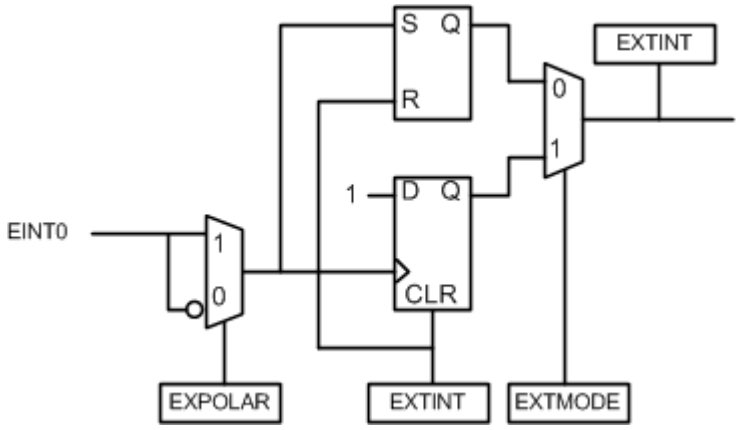
```
/*-------------------------------------------------------------------------
      Rotina de serviço às interrupções
*/
static int counter;

void isr() {
      U32 irq_status = io_read_u32(LPC210X_VIC + LPC2XXX_VICIRQStatus);
      if (irq_status & (1 << VIC_SOURCE_TIMER0)) {
            counter++;
            io_write_u32(LPC210X_T0 + LPC2XXX_TxIR, LPC2XXX_TxIR_MR1);
      }
}

int main() {
      interrupt_init();
      timer_init(1000);
      interrupt_unmask(VIC_SOURCE_TIMER0);
      while (1)
            ;
}
```

Interrupções externas

As fontes de interrupção são designadas por internas quando estão ligadas a periféricos internos e são designadas por externas quando são ligadas a periféricos externos.
Nos microcontroladores da família LPC2XXX os sinais de interrupção externos são tratados pela unidade System Control Block (SBC) antes de serem aplicados ao controlador de interrupções – VIC. No VIC as fontes de interrupção internas e externas são tratadas da mesma maneira.



Na unidade SCB existem três registos relacionados com as interrupções externas.
EXTINT – memoriza pedidos de interrupção; é necessário limpar no atendimento.
EXTWAKE – as interrupções podem acordar o processador do estado de power-down.
EXTMODE – a sensibilidade das entradas podem ser programadas a nível ou à transição.
EXTPOLAR – programação da sensibilidade ao nível zero ou um, ou, à transição descendente ou ascendente.
Os registos **EXTMODE** e **EXTPOLAR** só são programáveis na versão 01 do LPC2106. Nas versões anteriores estão sempre a zero o que implica que as entradas são sensíveis ao nível zero – é gerada interrupção se, e sempre que, uma destas entradas esteja a zero.

Exemplo

Incrementar uma variável a cada interrupção externa .
É apresentado apenas as o código diferente do exemplo anterior.

```
static void interrupt_init() {
      /* Inibir todas as interrupções */
      io_write_u32(LPC210X_VIC + LPC2XXX_VICIntEnClear, 0xffffffff);

      /* Limpar eventuais pedidos de interrupção via software */
      io_write_u32(LPC210X_VIC + LPC2XXX_VICSoftIntClear, 0xffffffff);

      /* Mapear a tabela de interrupções, residente no inicio da RAM,
            no endereço 0x0000000 */
      io_write_u32(LPC210X_SCB + LPC2XXX_MEMMAP, 2);

      /* Associar todas as fontes de interrupção à entrada IRQ */
      io_write_u32(LPC210X_VIC + LPC2XXX_VICIntSelect, 0);

      /* Conectar sinal EINT0 ao pino do circuito integrado */
      io_write_u32(LPC210X_PCB + LPC2XXX_PINSEL1, LPC2XXX_PINSEL_EINT0);

      /* Programar sensibilidade em modo edge-sensitive, só para LPC2106-01
      io_write_u32(LPC210X_SCB + LPC2XXX_EXTMODE,
            LPC2XXX_EXTxxx_INT0); */

      /* Programar sensibilidade em rising-edge sensitive, só para LPC2106-01
      io_write_u32(LPC210X_SCB + LPC2XXX_EXTPOLAR,
            LPC2XXX_EXTxxx_INT0); */

      /* Permitir que hajam interrupções ao nível do processador */
      interrupt_enable();
}


static int counter, counter_ext;

void isr() {
      U32 irq_status = io_read_u32(LPC210X_VIC + LPC2XXX_VICIRQStatus);
      if (irq_status & (1 << VIC_SOURCE_TIMER0)) {
            counter++;
            io_write_u32(LPC210X_T0 + LPC2XXX_TxIR, LPC2XXX_TxIR_MR1);
      }

      if (irq_status & (1 << VIC_SOURCE_EINT0)) {
            counter_ext++;

            /* Limpar o pedido de interrupção.
                  Mesmo com sensibilidade a nível,
                  depois do pedido desaparecer,
                  ainda fica assinalado neste registo */
            io_write_u32(LPC210X_SCB + LPC2XXX_EXTINT, LPC2XXX_EXTxxx_INT0);
      }

}

int main() {
      interrupt_init();
```

```
      timer_init(1000);
      interrupt_unmask(VIC_SOURCE_TIMER0);
      interrupt_unmask(VIC_SOURCE_EINT0);
      while (1)
            ;
}
```

Funcionamento em modo vectorizado

Funcionar em modo vectorizado significa saltar directamente para a ISR associada à entrada activada, sem ter que executar código auxiliar para selecção da rotina.
Este salto resulta do carregamento no PC, em articulação como o controlador de interrupções, do endereço da rotina de atendimento. A selecção do endereço é feita pelo controlador de interrupções em função da entrada activa.

Caso da família LPC2xxx

No controlador de interrupções da família **LPC2xxx** (VIC – Vectored Interrupt Controller) o registo que define o endereço da ISR é **VICVectAddr** com endereço absoluto **0xFFFFF030.**

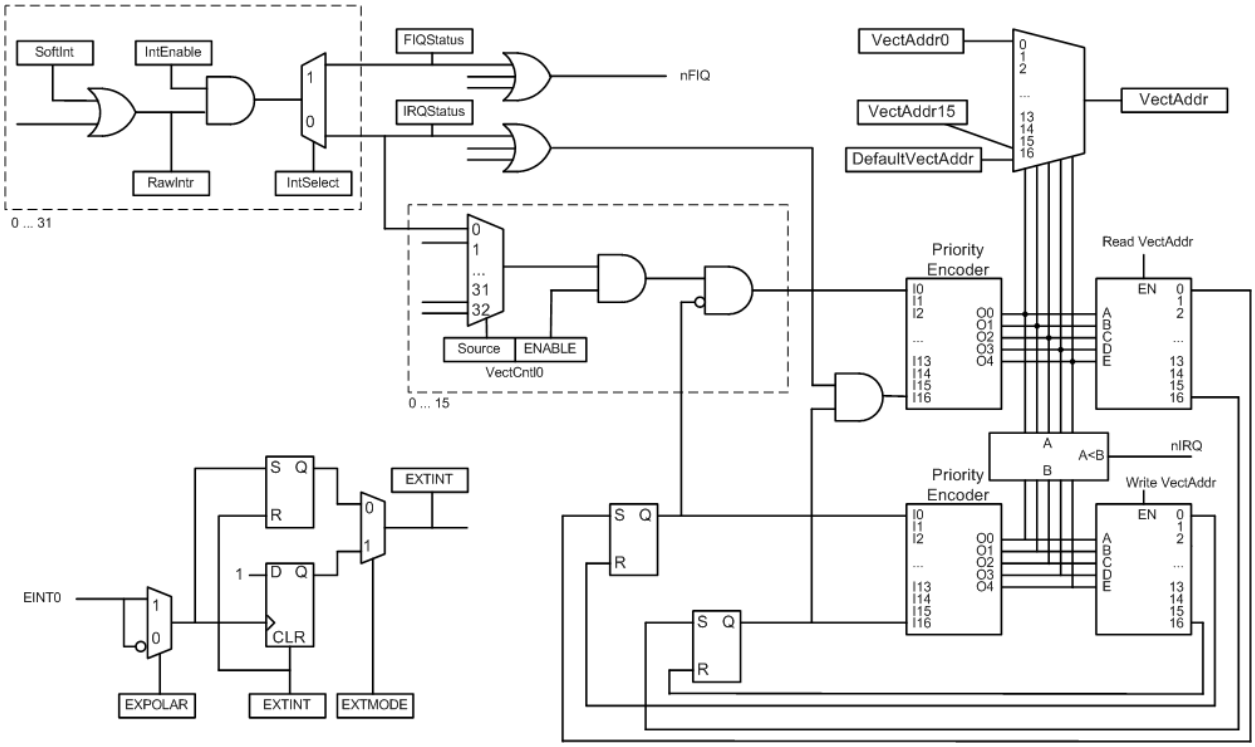
```
0x00000018 ldr        pc, [pc, #-0xFF0]
```

Esta instrução é executada como resposta à activação de IRQ e coloca em PC o conteúdo do registo **VICVectAddr**. O endereço deste registo, **0xFFFFF030**, é obtido subtraindo **0xFF0** a **0x00000020. 0x00000020** é o valor de PC quando a instrução **ldr pc, [pc, #-0xFF0]** é executada.

(No caso do controlador de interrupções ATMEL (AIC – Advanced Interrupt Controller) o registo em causa tem o endereço absoluto **0xFFFFF100**, por isso a instrução a usar é: **ldr pc, [pc, #-0xF20])**

Controlador de interrupções da família LPC2xxx

O controlador de interrupções tem por missão reunir os pedidos de interrupção dos periféricos e gerir os pedidos de interrupção ao processador.



As fontes associadas a IRQ podem ser vectorizadas, isto é, ser indicado o endereço da respectiva rotina de tratamento (registos **VICVectAddr0..15)**. As fontes vectorizadas têm associada uma prioridade que se indica nos registos **VICVectCntl0..15).**
As entradas não vectorizadas activam a rotina de serviço indicada em **VICDefVectAddr**. Esta rotina deve inquirir o controlador de interrupções, no registo **VICIRQStatus**, para determinar quais os periféricos que activaram o pedido e que devem ser servidos.

As fontes que activam a entrada FIQ não são vectorizaveis. No caso, pouco provável, de se associar mais do que uma fonte de interrupção à entrada FIQ, a rotina de serviço deve consultar **VICFIQStatus** para determinar qual foi o periférico que gerou o pedido.

O controlador de interrupções memoriza a prioridade da interrupção que está a ser servida e, activa IRQ, se uma fonte associada a uma maior prioridade for activada. Esta funcionalidade permite que rotinas associadas a uma menor prioridade sejam interrompidas para dar lugar ao atendimento de outras de maior prioridade.

Como consequência da memorização das prioridades que estão a ser servidas, o controlador de interrupções precisa de ser informado da terminação das rotinas de serviço. Esta informação é veiculada por uma escrita no registo **VICVectAddr** que indica, de cada vez que é realizada, a terminação do atendimento da interrupção de mais alta prioridade em processamento.

Ciclo de atendimento de interrupção

Um ciclo de atendimento de interrupção tem início no pedido por parte do periférico. Se a respectiva fonte não estiver inibida irá provocar a activação de IRQ. Em resposta o processador transfere o processamento para **0x00000018**. Neste local encontra-se uma instrução que carrega no PC o conteúdo do registo **VICVectAddr**. Este registo contém o endereço da rotina de

atendimento. Nesta altura o controlador regista "atendimento em curso" para o respectivo nível de prioridade.

Se a fonte de interrupção for vectorizada a rotina de serviço inicia de imediato o atendimento do periférico. Se for não vectorizada a rotina de serviço tem que determinar qual o periférico que originou o pedido.

Ao terminar, a rotina de atendimento informa o controlador para apagar o registo de "atendimento em curso". Se existir outra entrada activa (ou permanecer a mesma) o sinal IRQ mantém-se activado.

Como o atendimento decorreu com a *flag* I a 1 (desactivada) o processador não reage a um eventual pedido IRQ. Assim que o processador retomar ao programa interrompido e restabelecer a *flag* I a 0 (activada) inicia novamente um novo ciclo de atendimento.

Preparação

Como preparação para o atendimento de interrupções em modo vectorizado é necessário:

- Formar a tabela de excepções;

```
      .section ".vectors", "ax"

      ldr         pc, _reset_handler
      ldr         pc, _undefined_handler
      ldr         pc, _swi_handler
      ldr         pc, _prefetch_abort_handler
      ldr         pc, _data_abort_handler
      .word 0
      ldr         pc, [pc, #-0xFF0]
      ldr         pc, _fiq_handler

      _reset_handler:               .word reset_handler
      _undefined_handler:           .word undefined_handler
      _swi_handler:                 .word swi_handler
      _prefetch_abort_handler:      .word prefetch_abort_handler
      _data_abort_handler:          .word data_abort_handler
      _fiq_handler:                 .word fiq_handler
```

- Localizar a tabela de excepções;

```
MEMORY
{
      exception_table : ORIGIN = 0x40000000, LENGTH = 64
      ram : ORIGIN = 0x40000000 + 64, LENGTH = 0x10000 - 64 - 32
            /* 64 - tabela de excepções, 32 - usado pelo IAP */
}

SECTIONS
{
      .vectors : {
            *(.vectors*)
      } > exception_table

      ...
```

```
}
```

- Colocar a tabela de excepções visível;

```
io_write_u32(LPC210X_SCB + LPC2XXX_MEMMAP, 2);
```

- Associar a fonte de interrupção à entrada IRQ ou FIQ do processador;

```
io_write_u32(LPC210X_VIC + LPC2XXX_VICIntSelect, 0);
```

- Indicar o endereço da rotina de serviço à interrupção;

```
io_write_u32(LPC210X_VIC + LPC2XXX_VICVectAddr0, (U32)isr);
```

- Atribuir prioridade à fonte de interrupção no caso de ser fonte vectorizada;

```
io_write_u32(LPC210X_VIC + LPC2XXX_VICVectCntl0,
      (1 << 5) | VIC_SOURCE_EINT0);
```

- Desinibir o atendimento da fonte de interrupção (máscara);

```
io_write_u32(LPC210X_VIC + LPC2XXX_VICIntEnable,
      1 << VIC_SOURCE_EINT0);
```

- Desinibir o atendimento de interrupções no processador.

```
mrs   r0, cpsr
and   r0, r0, #~(1 << 7)
msr   cpsr_c, r0
```

Durante a depuração de código pode acontecer reiniciar-se o programa sucessivamente sem se fazer *reset* ao *hardware*. Nestas circunstâncias, o controlador de interrupções permanece num qualquer estado anterior.

O seguinte troço de código envia uma sequência de EOI que limpa eventuais marcações de rotinas de serviço em execução.

```
for (i = 0; i < 16; ++i)
      io_write_u32(LPC210X_VIC + LPC2XXX_VICVectAddr, 0);
```

No caso das interrupções externas é necessário também:

- Programar a sensibilidade das entradas (quando aplicável);

```
io_write_u32(LPC210X_SCB + LPC2XXX_EXTMODE,
      LPC2XXX_EXTMODE0_EDGE | LPC2XXX_EXTMODE1_EDGE);

io_write_u32(LPC210X_SCB + LPC2XXX_EXTPOLAR,
      LPC2XXX_EXTPOLAR0_FALLING | LPC2XXX_EXTPOLAR1_FALLING);
```

- Associar o pino do circuito integrado à função de interrupção externa;

```
io_write_u32(LPC210X_PCB + LPC2XXX_PINSEL1,
      LPC2XXX_PINSEL_EINT0);
```

Para garantir que não há pedidos pendentes em **EXTINT** que possam falsear as experiências deve limpar-se este registo.

```
io_write_u32(LPC210X_SCB + LPC2XXX_EXTINT, 0xf);
```

Interrupções falsas

As falsas interrupções podem ocorrer se a fonte que originou a interrupção for desactivada depois do CPU ter iniciado o processo de atendimento da interrupção (IRQ ou FIQ) e antes do registo **VICVectAddr** ter sido lido. Nesta situação o processador continua com o processo mas o endereço recolhido em **VICVectAddr** não é o correspondente à fonte que gerou a interrupção mas o endereço que se encontra em **VICDefVectAddr**. Este registo é posto a zero em *reset*, por isso se não for alterado as falsas interrupções provocam a excepção *reset*.
Se não existirem atendimentos não vectorizados a rotina de atendimento de falsas interrupções não deve fazer mais do que sinalizar o fim do atendimento da interrupção e retornar.

Aninhamento de Interrupções

O aninhamento de interrupções acontece quando o processador atende uma interrupção subsequente antes de concluir o processamento da anterior. Este procedimento aplica o critério de prioridades ao ponto de um interrupção de maior prioridade interromper o processamento de uma interrupção de menor prioridade.

Na implementação deste método, o controlador de interrupções memoriza a prioridade da interrupção que está a ser atendida, e precisa de ser informado quando o atendimento termina.

Se durante um atendimento surgir um pedido de maior prioridade o controlador activa o sinal do processador (IRQ). Isso não acontecerá enquanto o mesmo pedido permanecer activo ou os pedidos existentes forem de prioridade inferior.

Para se usar esta característica do controlador de interrupções cabe ao software a preparação do ambiente de execução para a interrupção aninhada possa suceder.

Na arquitectura ARM, para que uma interrupção seja atendida é necessário que a flag I do processador esteja activa e o registo LR do modo IRQ esteja disponível, isto é, o processador esteja a executar num modo diferente de IRQ. Modo utilizador, por exemplo.

Uma estratégia possível é a seguinte: começa-se por salvar o contexto do programa interrompido – **`r0`** a **`r3, ip`**, **`lr_irq`**, tal como anteriormente, e preserva-se também o modo do processador que se encontra em **`spsr_irq`**. Depois comuta-se para o esse modo, activa-se a flag I (**`msr cpsr_c, #0x1f)`** e prossegue-se com o atendimento usando o stack do programa interrompido.

Se durante o atendimento ocorrer outra interrupção, o processo reinicia-se subindo os níveis de ocupação dos *stacks* tanto do modo IRQ como do modo de utilizador.

```
      .text

      .global isr0_asm
isr0_asm:
      sub   lr, lr, #4
      stmdb sp!, {r0 - r4, ip, lr}
      mrs   r4, spsr
      orr   r4, r4, #(1 << 7)
      msr   cpsr, r4
      str   lr, [sp, #-4]!

      bl    isr0

      ldr   lr, [sp], #4
      msr   cpsr_c, #0xd2
      msr   spsr, r4
      ldmia sp!, {r0 - r4, ip, pc}^
```

Note-se que ao comutar para modo utilizador, vão ser expostos os registos **`sp`** e **`lr`** do programa original, estes registos vão naturalmente ser utilizados durante o processamento. Quanto ao **`sp`** não há problema, pela sua natureza pode continuar a ser usado, quanto a **`lr`** este precisa de ser salvo antes de se iniciar a chamada a outras funções.

Este método requer que os *stacks*, de modo utilizador e de modo IRQ, tenham capacidade para armazenar os dados produzidos nas sucessivas entradas em atendimento de interrupção.

Partilha de linhas de interrupção

A partilha de entradas de interrupção por vários periféricos é uma solução usada quando o número de entradas de interrupção é insuficiente.

No caso de entradas sensíveis à transição é necessário garantir que a entrada de interrupção é desactivada durante o atendimento. Se isso não acontecer e a entrada de interrupção permanecer activa não haverá nova transição e o controlador de interrupções não apresentará novos pedidos de interrupção ao processador.

Este processo só poderá ser usado se, no processamento da interrupção, se poder garantir que todas as fontes de interrupção foram processadas, e consequentemente desactivadas, habilitando o controlador a gerar novos pedidos.

API para processamento de interrupções

A seguinte interface de programação reúne as funcionalidades básicas oferecidas por um sistema de atendimento de interrupções. A sua utilização torna mais cómoda a utilização deste mecanismo e potencia a reutilização de código.

`void interrupt_init();`
Iniciar o sistema de atendimento de interrupções, tanto ao nível do processador como do controlador de interrupções.

`void interrupt_set_isr(int number, void (*isr)(void*), void * obj);`
Definir a rotina de atendimento de interrupção para uma dada entrada de interrupção.

`void interrupt_mask(int number);`
Inibir o atendimento de determinada entrada de interrupção mesmo se activa.

`void interrupt_unmask(int number);`
Permitir o atendimento de determinada entrada de interrupção.

`void interrupt_end();`
Elimina, no controlador de interrupções, o registo "atendimento em curso" de maior prioridade que estiver activo.

`void interrupt_configure(int number, int edge, int direction);`
Configurar a sensibilidade da entrada de interrupção (nível e sentido).

`void interrupt_priority(int number, int priority);`
Atribui um nível de prioridade a uma entrada de interrupção.

`interrupt_enable();`
Permitir o atendimento de interrupções por parte do processador.

`interrupt_disable();`
Inibir o atendimento de interrupções por parte do processador.

`int interrupt_save_and_disable();`
Inibir o atendimento de interrupções por parte do processador guardando o estado anterior.

`interrupt_restore(int);`
Repor um estado de permissão de atendimento de interrupções.

Exemplo de utilização da API

Atendimento de pedidos de interrupção na entrada externa ENIT0.

A função **`interrupt_end`** normalmente é invocada no final da rotina de interrupção. No entanto, o programa de utilização pode invocá-la em qualquer altura sabendo que a partir daí o controlador de interrupções pode apresentar outros pedidos, incluindo de mais baixa prioridade.

```
static void isr() {
      io_write_u32(LPC210X_SCB + LPC2XXX_EXTINT, LPC2XXX_EXTINT_EINT0);
      ++count;
      interrupt_end();
}
```

As funções **`interrupt_set_isr`** e **`interrupt_unmask`** devem ser invocadas durante a iniciação de periféricos.

```
static void peripheral_init() {
      io_write_u32(LPC210X_SCB + LPC2XXX_EXTINT, LPC2XXX_EXTINT_EINT0);
      interrupt_set_isr(VIC_SOURCE_EINT0, isr, 0);
      interrupt_unmask(VIC_SOURCE_EINT0);
}
```

A função **`interrupt_init`** deve ser invocada uma única vez na fase de inicialização do sistema. A permissão geral de atendimento de interrupções **`interrupt_enable`** deve ser executada também nesta fase. Numa altura em que já não se corre o risco de receber a interrupção de um periférico ainda não iniciado.

```
int main() {
      ...
      interrupt_init();
      peripheral_init();
      interrupt_enable();
      while (1) {
            ...
      }
      return 0;
}
```

Durante a operação do programa pode ser necessário inibir o atendimento de interrupções, quer relativos a um periférico específico, caso em que se deve usar o par de funções `interrupt_mask/interrupt_unmask`, quer para todo o sistema, caso em que se deve usar `interrupt_save_disable/interrupt_restore`.

Implementação da API no LPC2106

Apresenta-se abaixo uma implementação simplificada da API. Esta não tira partido do mecanismo de prioridades do controlador de interrupções e como consequência não permite o atendimento aninhado de interrupções.

```
typedef void (*Isr)(void*);

static struct { void * obj; Isr isr; } irq_tab[32];

static int spurious_counter = 0;

void isr() {
      /* Ler interrupções activas */
      U32 irq_stat = io_read_u32(LPC210X_VIC + LPC2XXX_VICIRQStatus);

      /* Identificar interrupção activa .
      Esta pesquisa dá prioridade aos números mais baixos */
      int irq_num = 0;
      while ((irq_stat & 1) == 0) {
            irq_stat >>= 1;
            irq_num++;
      }
      /* Chamar a rotina de serviço do utilizador */
      if (irq_num < 32) {
            assert(irq_tab[irq_num].isr != 0);
            irq_tab[irq_num].isr(irq_tab[irq_num].obj);
      }
      else
            spurious_counter++;
}

void interrupt_end() {
      /* Assinalar o fim do atendimento da interrupção */
      io_write_u32(LPC210X_VIC + LPC2XXX_VICVectAddr, 0);
```

```
}

      .text
      .global     isr_asm
isr_asm:
      sub         lr, lr, #4
      stmdb       sp!, {r0 - r3, ip, lr}
      bl          isr
      ldmia       sp!, {r0 - r3, ip, pc}^

void interrupt_init() {
      int i;

      /* Inibir todas as interrupções */
      io_write_u32(LPC210X_VIC + LPC2XXX_VICIntEnClr, 0xffffffff);

      /* Mapear a tabela de interrupções, residente no inicio da RAM,
            no endereço 0x0000000 */
      io_write_u32(LPC210X_SCB + LPC2XXX_MEMMAP, 2);

      /* Limpar logica de prioridades
            (devido a testes anteriores sem reset) */
      for (i = 0; i < 2; ++i)
            io_write_u32(LPC210X_VIC + LPC2XXX_VICVectAddr, 0);

      /* Limpar eventuais pedidos de interrupções externas pendentes
            (devido a testes anteriores sem reset) */
      io_write_u32(LPC210X_SCB + LPC2XXX_EXTINT, 0x7);

      /* Desactiva lógica de vectorização */
      for (i = 0; i < 16; ++i)
            io_write_u32(LPC210X_VIC + LPC2XXX_VICVectCntl0 + i*4, 0);

      /* Associar todas as fontes de interrupção à entrada IRQ */
      io_write_u32(LPC210X_VIC + LPC2XXX_VICIntSelect, 0);

      /* Indicar a primeira instância da
            rotina de atendimento de interrupção */
      io_write_u32(LPC210X_VIC + LPC2XXX_VICDefVectAddr, (U32)isr_asm);

      /* Limpar tabela de rotinas de interrupção do utilizador */
      for (i = 0; i < sizeof(irq_tab)/sizeof(irq_tab[0]); ++i)
            irq_tab[i].isr = 0;
}

void interrupt_set_isr(int irq, Isr isr, void * obj) {
     irq_tab[irq].obj = obj;
     irq_tab[irq].isr = isr;
}

void interrupt_mask(int irq) {
      io_write_u32(LPC210X_VIC + LPC2XXX_VICIntEnClr, 1 << irq);
}

void interrupt_unmask(int irq) {
      io_write_u32(LPC210X_VIC + LPC2XXX_VICIntEnable, 1 << irq);
}
```

```
void interrupt_configure(int irq, int edge, int direction) {
      /* Só é possível fazer esta programação na versão 1 do LPC2106 */
}

void interrupt_enable() {
      asm("mrs    r0, cpsr");
      asm("bic    r0, r0, #(3 << 6)");
      asm("msr    cpsr_c, r0");
}

void interrupt_disable() {
      asm("mrs    r0, cpsr");
      asm("orr    r0, r0, #(3 << 6)");
      asm("msr    cpsr_c, r0");
}
```

Nesta implementação é desaproveitada a lógica de priorização e vectorização, sendo invocada sempre a mesma rotina de tratamento da interrupção, independentemente da entrada de interrupção activada. Esta rotina, por pesquisa no registo **`VICIRQStatus`**, determina qual a entrada activa. O algoritmo de pesquisa determina o critério de prioridade.

Referências

ARM System-on-chip architecture – capítulo 5
UM10275 - LPC2104/2105/2106 User manual
AN10381 - Nesting of interrupts in the LPC2000
AN10414 - Handling of spurious interrupts in the LPC2000
Projecto de Sistemas Digitais, Pimenta Rodrigues, capítulo 14