Microcontrolador LPC2106

Table of Contents

LPC2106

O microcontrolador LPC2106 é constituido por um conjunto de periféricos em torno do processador ARM7TDMI-S. Os periféricos interligam ao processador através de barramentos especializados.

Periféricos

Memória interna, 64K de RAM estática e 128K de FLASH.
Controlador de interrupções.
Dois UARTs.
Interfaces de comunicação I2C e SPI..
Dois temporizadores.
Unidade PWM com 6 saídas.
Calendário.
Temporizador Watchdog.
Pinos de entrada/saída de utilização genérica.

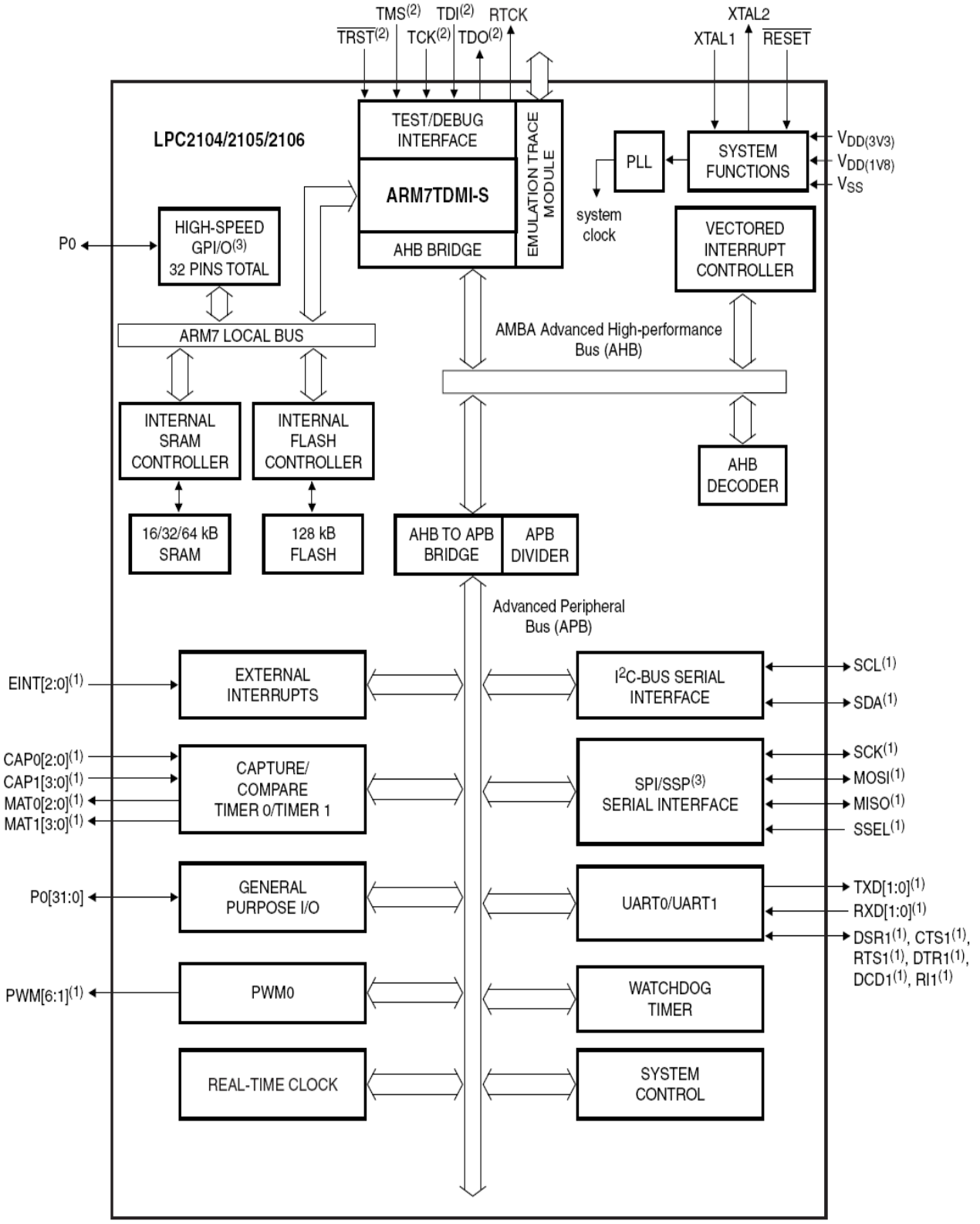
Meios de controlo do consumo de energia

Possibilidade de activar/desactivar periféricos.
Dois modos de baixo consumo – idle e power down.
Controlo da frequência de trabalho através de PLL.

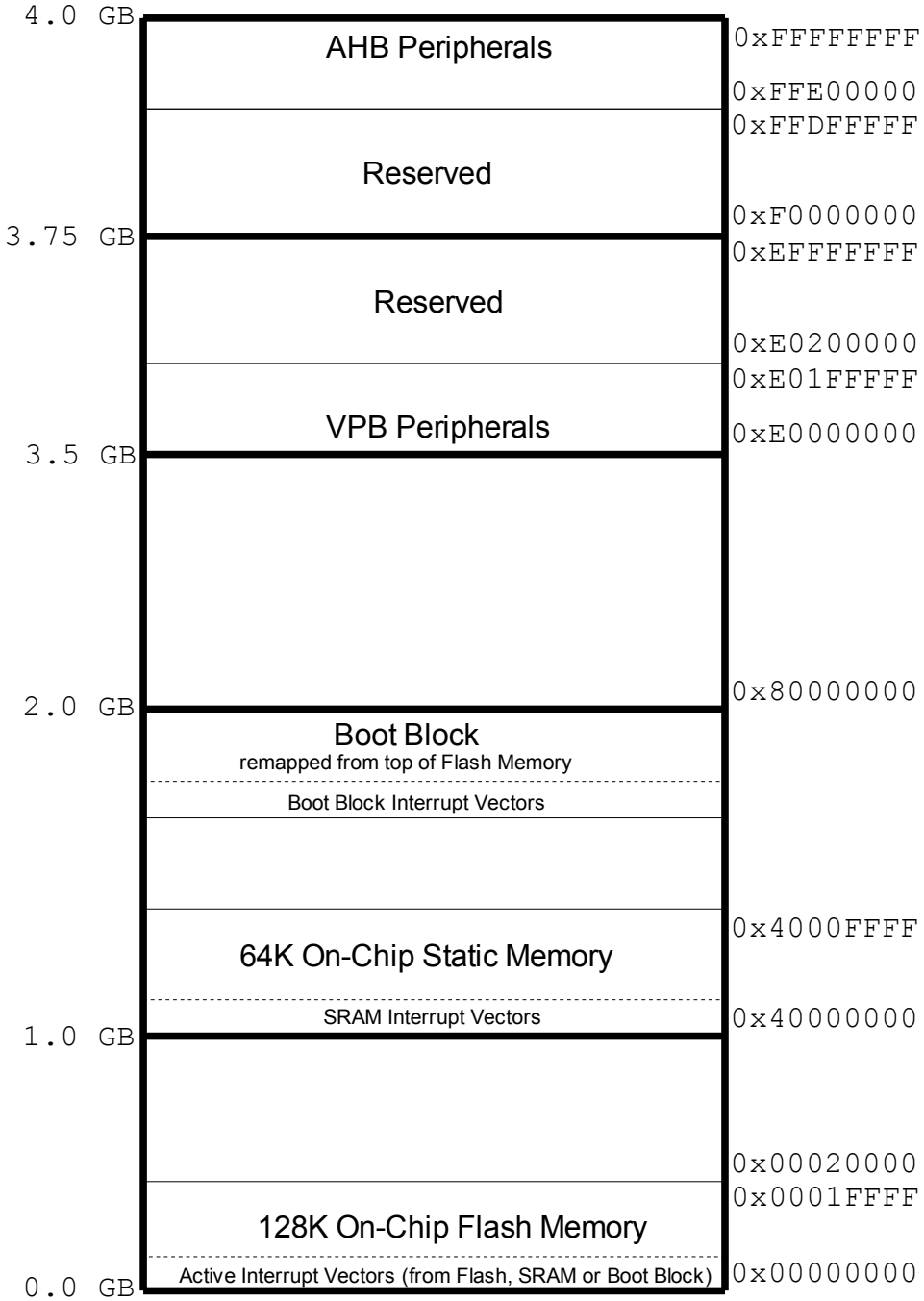
Meios de debug por hardware

Standard ARM Test/Debug interface.
RealMonitor.
Emulation Trace.

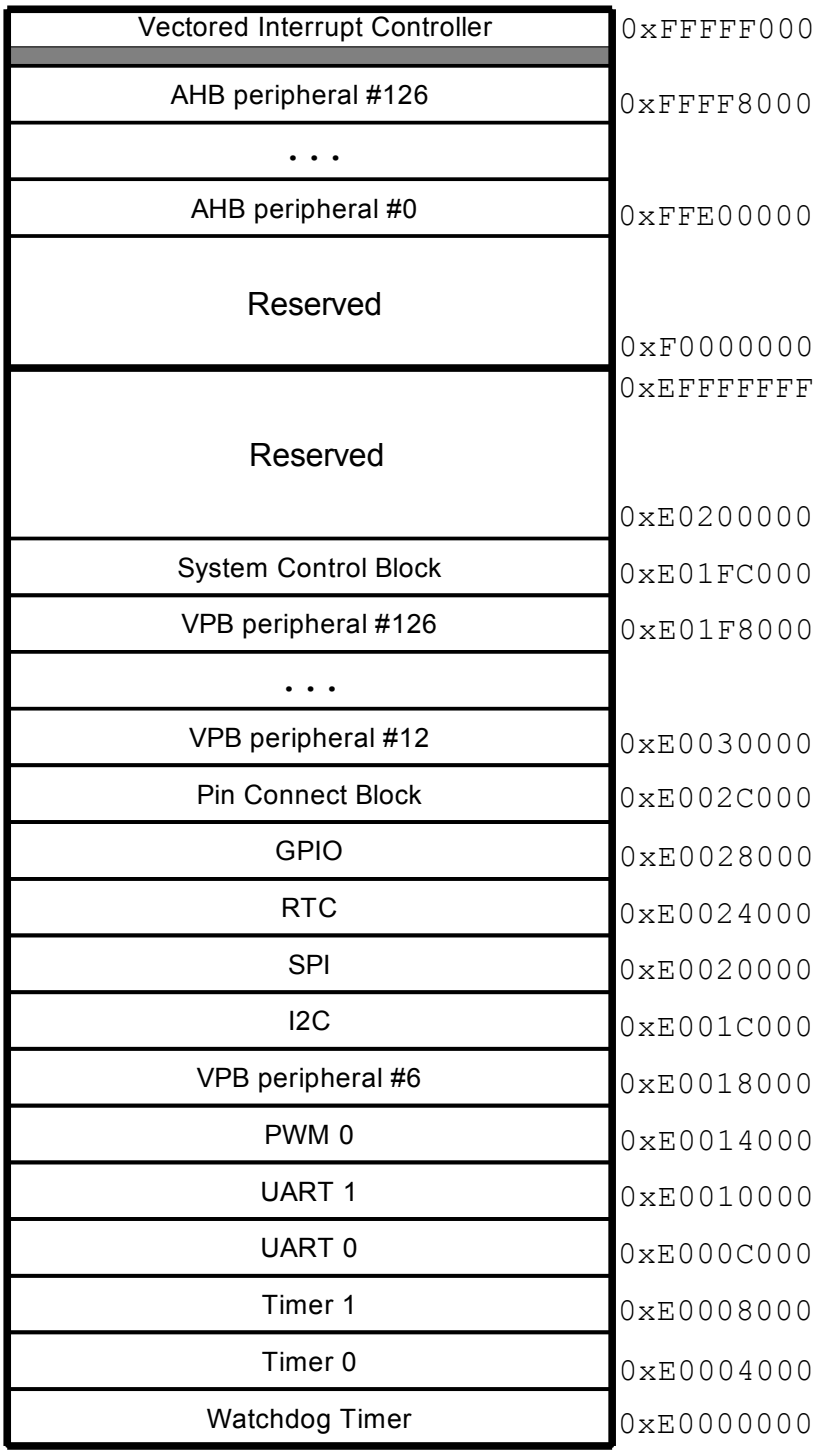
Diagrama de blocos



Mapa de memória



Mapeamento dos periféricos



O espaço de memória reservado a periféricos é repartido pelos barramentos AHB e VPB. A gama 0xF0000000 a 0xFFFFFFFF dá acesso aos periféricos do barramento AHB que contém apenas o controlador de interrupções (VIC), a gama 0xE0000000 a 0xEFFFFFFF dá acesso ao barramento VPB onde estão ligados os restantes periféricos. Os espaços de memória reservados aos periféricos é divididos em blocos de 16 KByte e é atribuído um bloco a cada periférico.

Bloco de controlo do sistema

O bloco de controlo do sistema engloba um conjunto de funcionalidades que dizem respeito ao funcionamento geral do microcontrolador. Fazem parte deste bloco as seguintes componentes:

- oscilador principal;
- entradas de interrupção externas;
- mapeamento de memória;
- PLL;
- consumo de energia;
- frequência dos periféricos;
- modo de funcionamento do processador.

Relógio do sistema

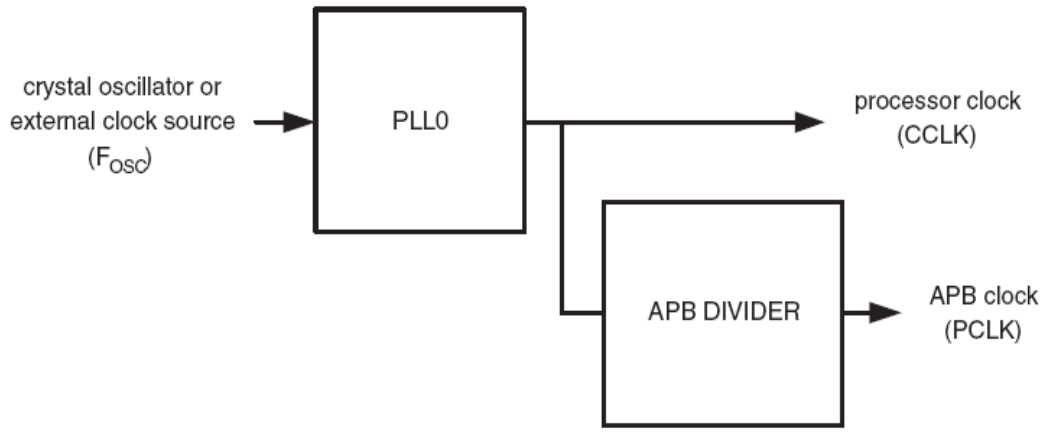
Existe um oscilador principal baseado em cristal ou numa fonte externa que opera na gama 10MHz a 25 MHz. O sinal gerado por este oscilador designa-se **Fosc**. No caso do LPC-H2106 da Olimex o seu valor é 14745600 Hz.

A frequência máxima do processador LPC2106 é 60 MHz. Após o reset, o PLL encontra-se desactivado e o processador opera directamente com o sinal do oscilador (CCLK = Fosc).

Relógio dos periféricos

Os periféricos operam com o sinal de relógio **PCLK**. Este sinal é derivado de CCLK e é programável como igual, metade ou um quarto de CCLK. A programação após o reset é de um quarto.

No caso do LPC-H2106 da Olimex o seu valor é 14745600 / 4 = 3686400 Hz.



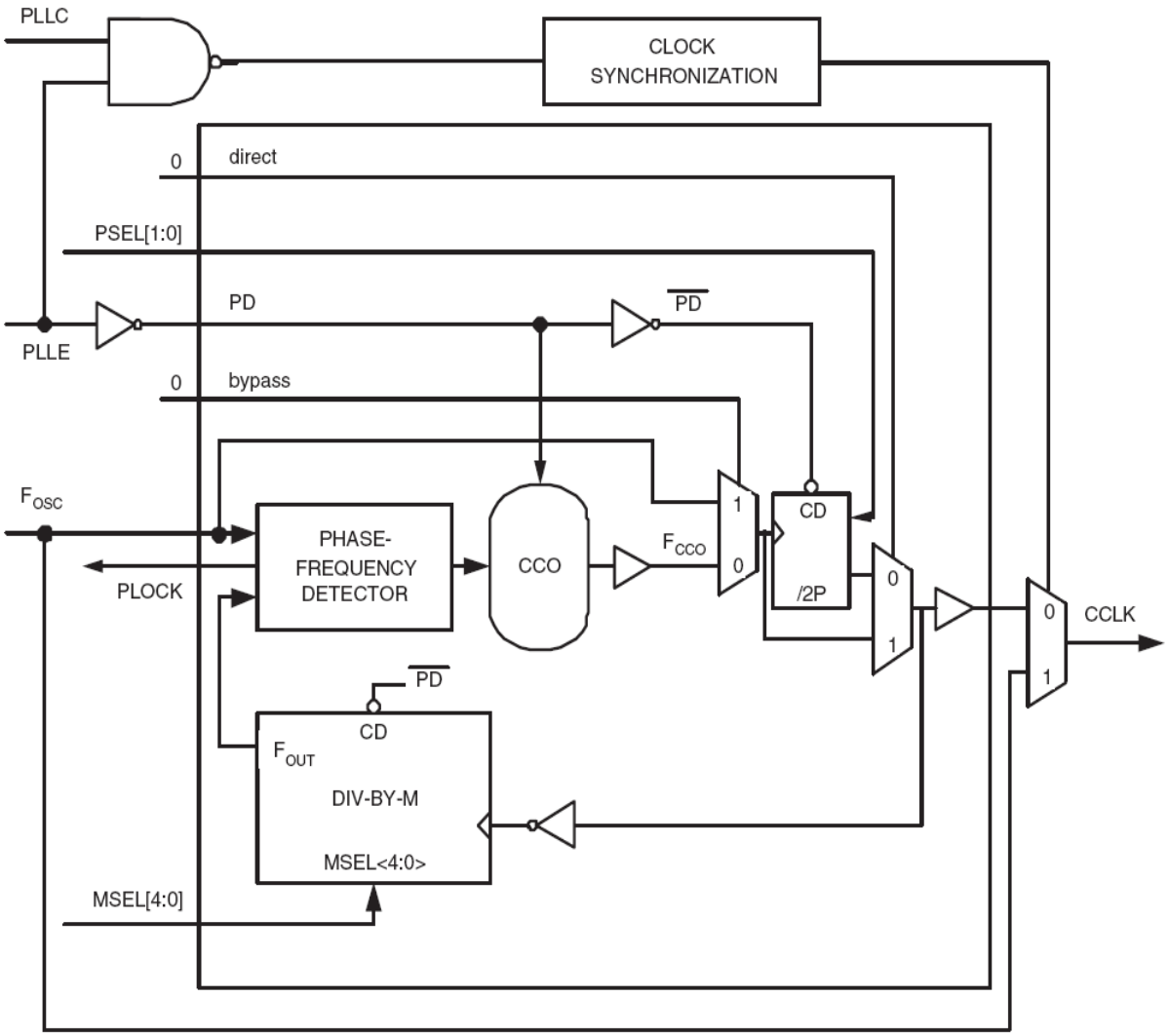
Interface de programação

VPB clock rate, 0xE01FC100, VPBDIV, RW

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
																														0	0

VPBDIV7/35
00 - ¼
01 - 1
10 - ½

PLL



O bloco PLL permite gerar um sinal de relógio de diversas frequências a partir do oscilador principal. A definição da frequência de saída é feita através dos parâmetros M e P que relacionam CCLK com Fosc da seguinte forma:

CCLK = M * Fosc

Fcco é um relógio interno do PLL, o seu valor é:

Fcco = Fosc * M * 2 * P

Os intervalos de funcionamento de cada um dos relógios são:

Fosc – 10 MHz a 25 MHz

CCLK – 10 MHz a 60 MHz

Fcco – 156 MHz a 320 MHz

No caso da placa LPC-H2106 da Olimex, com cristal de 14745600 Hz, e para obter uma frequência próxima de 60 MHz para o processador - CCLK, podem usar-se o valor 4 no parâmetro M e o valor 2 no parâmetro P.

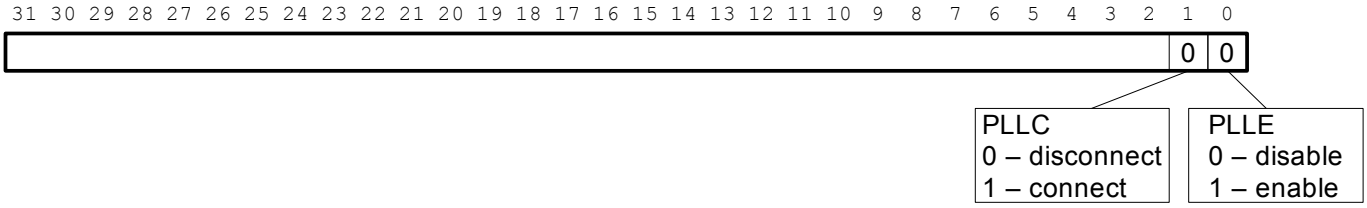
Fosc = 14745600 Hz

CCLK = 147456 00 x 4 = 58982400 Hz

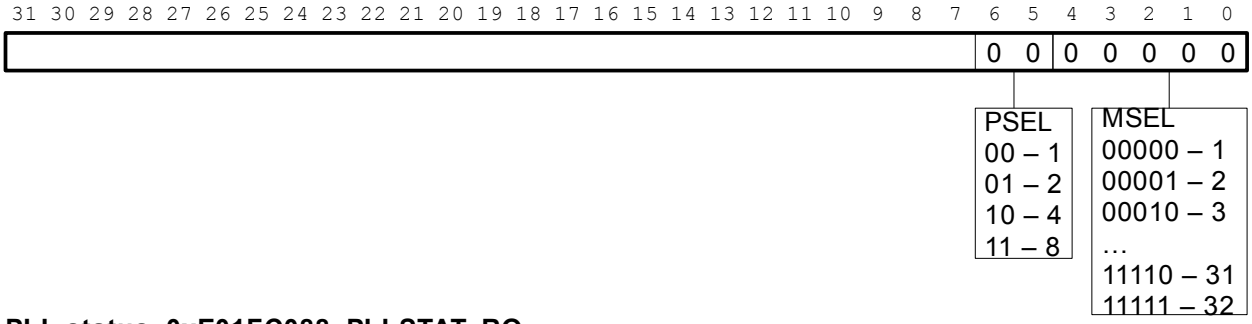
Fcco = 14745600 x 4 x 2 x 2 = 235929600 Hz

Registos de controlo do PLL

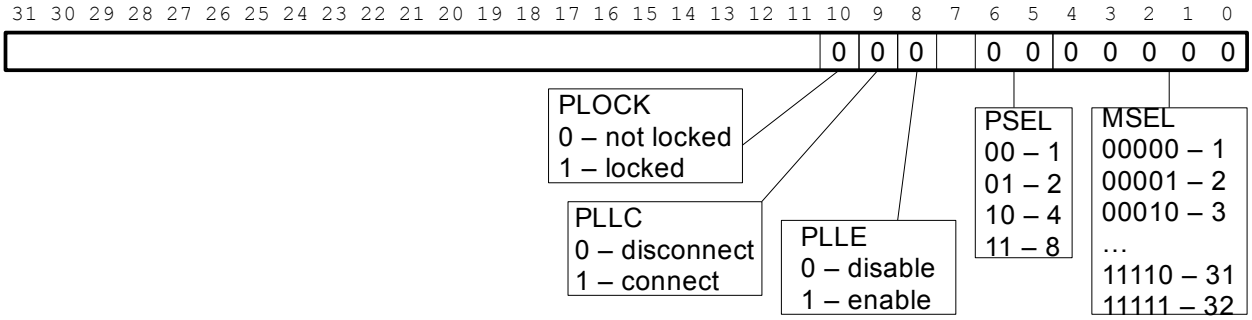
PLL control, 0xE01FC080, PLLCON, RW



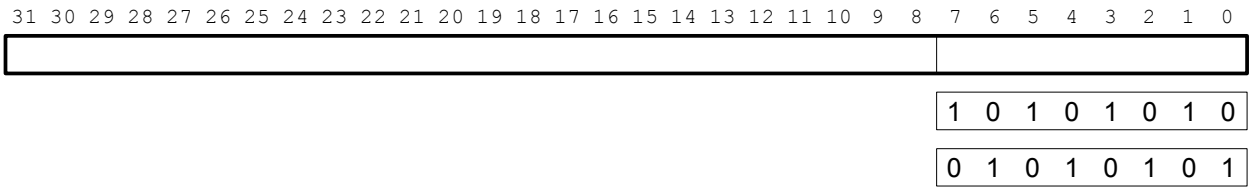
PLL configuration, 0xE01FC084, PLLCFG, RW



PLL status, 0xE01FC088, PLLSTAT, RO



PLL feed, 0xE01FC08C, PLLFEED, WO



Exemplo de programação do PLL

```
io_write_u32(LPC210X_SCB + LPC2XXX_PLLCFG,  0x23);
io_write_u32(LPC210X_SCB + LPC2XXX_PLLCON,  0x01);
io_write_u32(LPC210X_SCB + LPC2XXX_PLLFEED, 0xaa);
io_write_u32(LPC210X_SCB + LPC2XXX_PLLFEED, 0x55);
do {
     io_read_u32(LPC210X_SCB + LPC2XXX_PLLSTAT, aux);
} while (aux & (1 << 10));
io_write_u32(LPC210X_SCB + LPC2XXX_PLLCON,  0x03);
```

```
io_write_u32(LPC210X_SCB + LPC2XXX_PLLFEED, 0xaa);
io_write_u32(LPC210X_SCB + LPC2XXX_PLLFEED, 0x55);
```

Consumo de energia

O sistema tem dois modos de funcionamento em baixo consumo: *idle* e *power down.*

No modo *idle* o processador pára de executar instruções mas os periféricos mantêm a actividade e podem gerar interrupções. No modo *idle*, o sistema elimina o consumo do processador, o consumo da memória, dos respectivos controladores e dos barramentos.

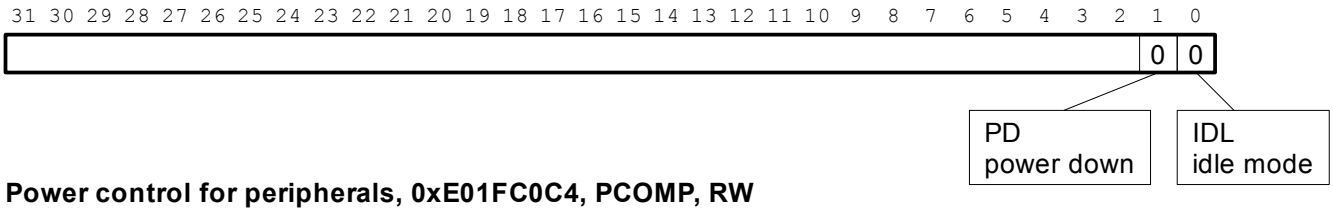
No modo *power down* o oscilador é desligado e como consequência o processador e periféricos que necessitem de relógio deixam de funcionar. Apenas os periféricos que não necessitam de relógio continuam em funcionamento, o que se resume ao controlador de interrupções. Os valores dos registos do processador, periféricos e memória RAM são preservados.

O sistema pode regressar de novo ao modo de funcionamento normal – *wakeup*, através da activação de uma interrupção. O processador retoma o processamento normal, exactamente no ponto onde tinha sido suspenso, sem execução incorrecta ou incompleta de instruções.

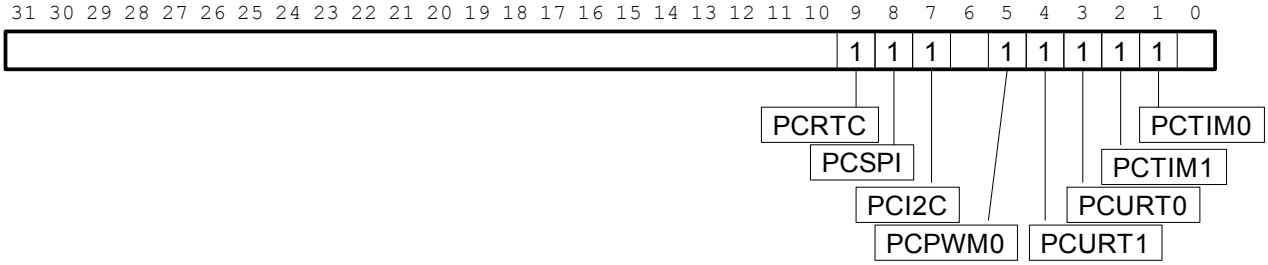
Em operação normal, os periféricos que não estão a ser usados podem ser desactivados. Após o *reset* , por omissão, todos os periféricos estão activados.

Interface de programação

Power control, 0xE01FC0C0, PCON, RW



Power control for peripherals, 0xE01FC0C4, PCOMP, RW



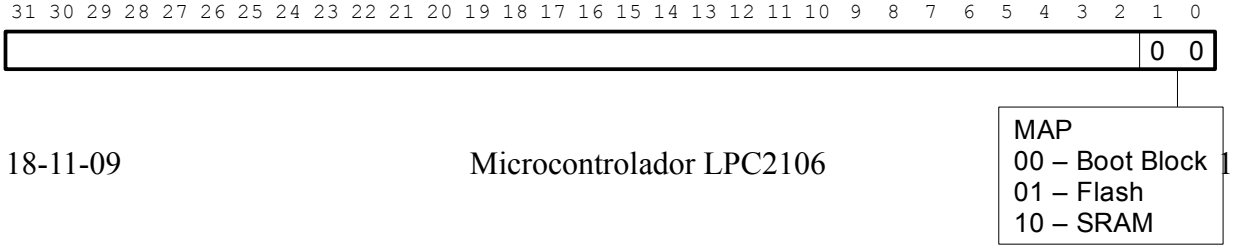
Exemplo de programação

```
U32 aux = io_read_u32(LPC210X_SCB + LPC2XXX_PCONP);
io_write_u32(LPC210X_SCB + LPC2XXX_PCONP, aux | LPC2XXX_PCONP_SPI);
```

Mapeamento da memória

Os endereços de 0x00000000 a 0x1000003F estão naturalmente associados ao início da memória Flash. No entanto, após o *reset*, surge mapeada nestes endereços a base do Boot Block. Por programação pode-se mudar para a RAM ou para o início da memória Flash.

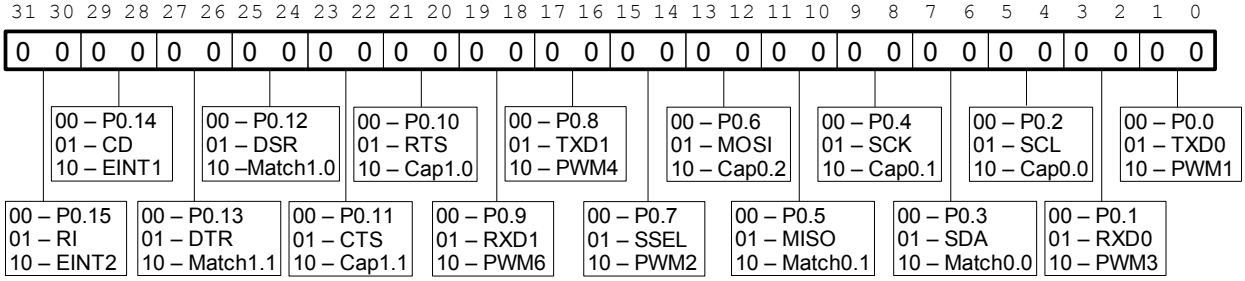
Memory mapping control, 0xE01FC040, MEMMAP, RW



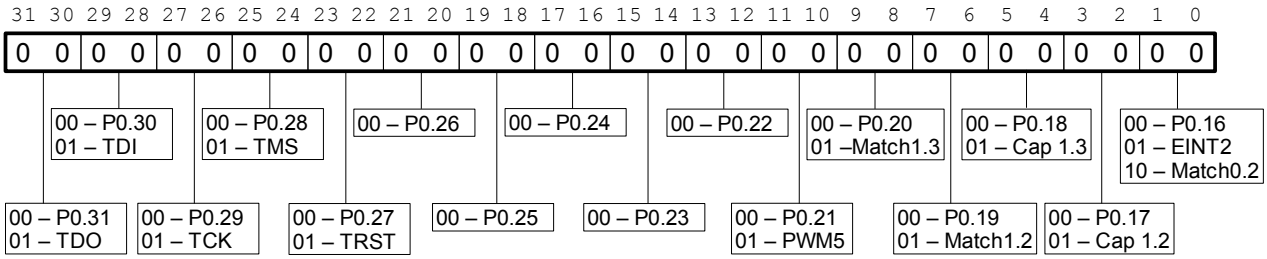
General Purpose Input/Output

Para economia de pinos no empacotamento do microcontrolador a generalidade dos pinos são partilhados por duas ou três funções. O periférico Pin Connect Block permite definir a função associada a cada pino. No *reset* todos os pinos ficam associados ao GPIO, salvo se se entrar em modo *debug* (DBGSEL a VCC) em que os pinos 22 a 36 são associados a funções de *debug* inalteráveis.

Pin Function Select Register 0, 0xE002C000, PINSEL0, RW



Pin Function Select Register 1, 0xE002C004, PINSEL1, RW



Exemplo de programação

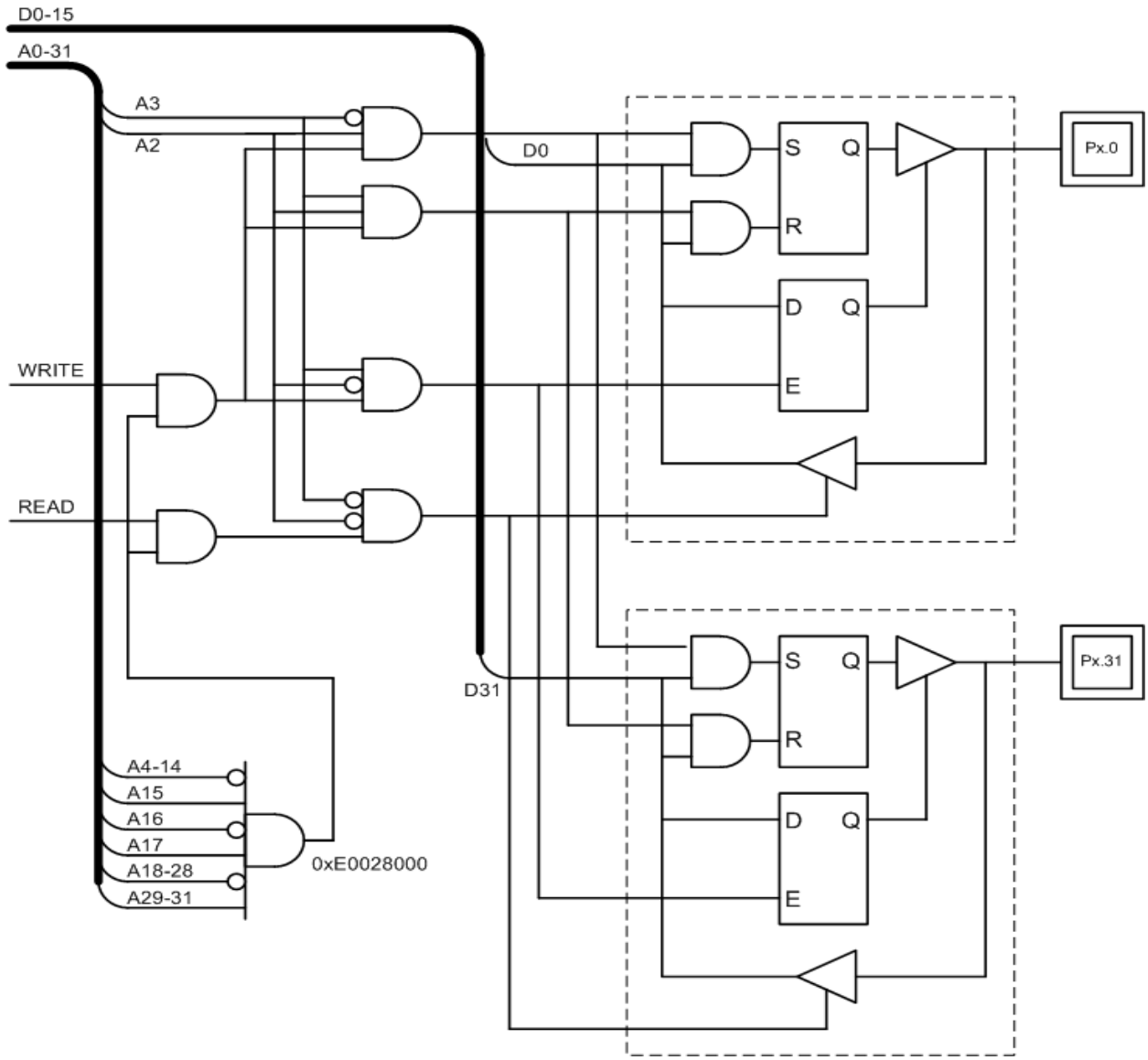
Ligar os pinos 13 e 14 do LPC2106 aos sinais TXD0 e RXD0 do UART0.

```
io_write_u32(LPC210X_PIN + LPC2XXX_PINSEL0, (1 << 2) | (1 << 0));
```

Ligar o pino 37 do LPC2106 à saída MAT1.0 do Timer 1.

```
io_write_u32(LPC210X_PIN + LPC2XXX_PINSEL0, 2 << 24);
```

Os pinos do porto de entradas/saídas genérico (GPIO) são programáveis para entrada ou para saída individualmente. A saída de dados é feita sobre um *flip-flop* tipo Set-Reset o que permite a utilização de bits diferentes em contextos independentes. Já a programação dos pinos como entrada ou como saída é feita sobre um *flip-flop* tipo D, logo, não é possível definir a característica entrada/saída de um pino independentemente dos outros.



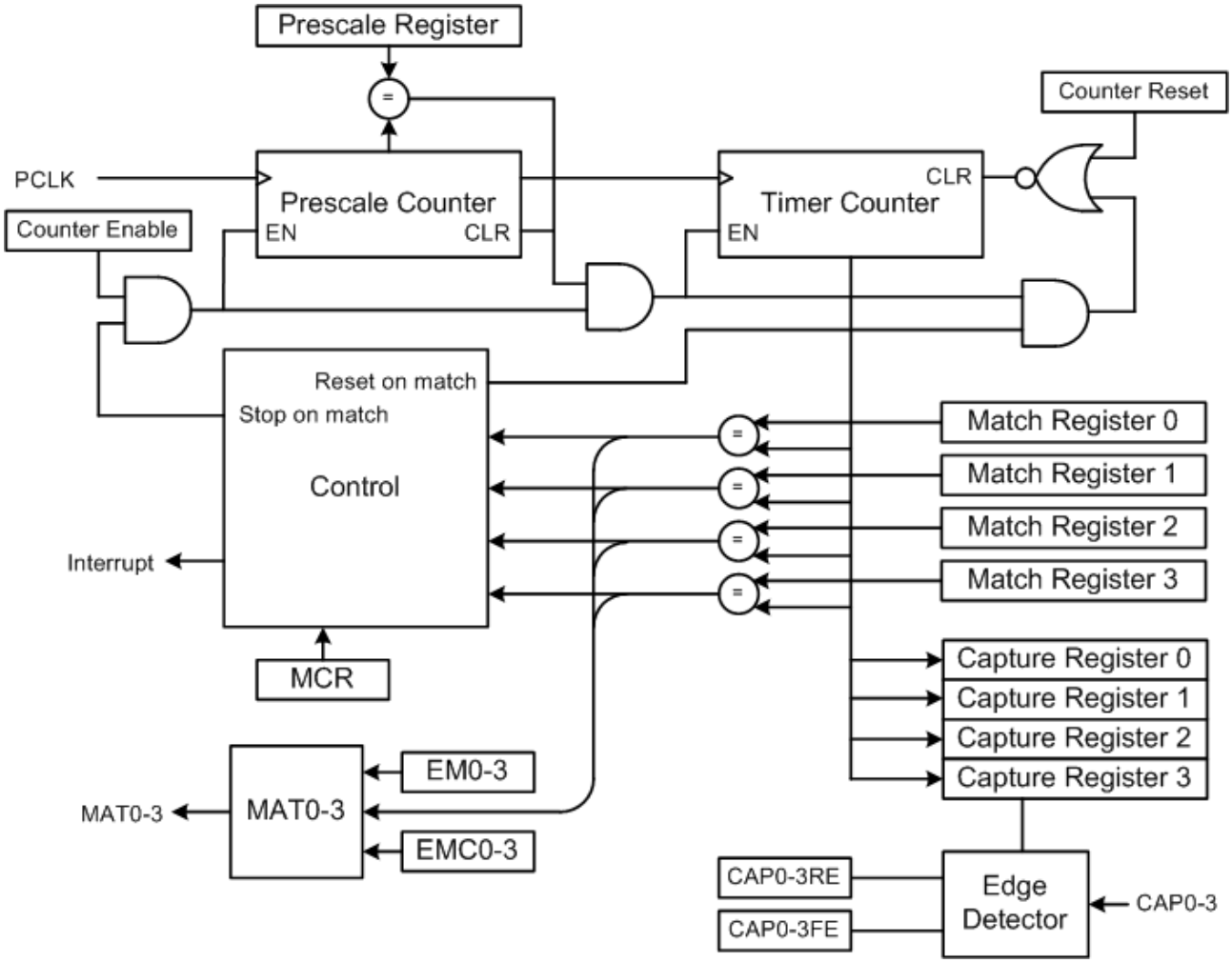
Exemplo de programação

Admitindo um LED ligado no pino P0.16 e um botão no pino P0.2, reflectir no LED o estado do botão.

```
WRITE_U32(LPC210X_GPIO + LPC2XXX_IODIR, 1 << 16);
while (1) {
    U32 aux;
    io_read_u32(LPC210X_GPIO + LPC2XXX_IOPIN, aux);
    if (aux & (1 << 2))
        io_write_u32(LPC210X_GPIO + LPC2XXX_IOSET, 1 << 16);
    else
        io_write_u32(LPC210X_GPIO + LPC2XXX_IOCLR, 1 << 16);
}
```

Timer

O timer é baseado em dois contadores encadeados, o Prescale Counter (PC) e o Timer Counter (TC). Ambos executam contagens crescentes. O PC é constantemente comparado com o Prescale Register (PR), quando é igual incrementa o TC e coloca PC a zero. Esta montagem permite dividir o PCLK por qualquer valor e incrementar o TC a qualquer ritmo (inferior a PCLK).



Interface de programação

IR	Interrupt Register	R/W	0	0x000
TCR	Timer Control Register	R/W	0	0x004
TC	Timer Counter	R/W	0	0x008
PR	Prescale Register	R/W	0	0x00C
PC	Prescale Counter	R/W	0	0x010
MCR	Match Control Register	R/W	0	0x014
MR0	Match Register 0	R/W	0	0x018
MR1	Match Register 1	R/W	0	0x01C
MR2	Match Register 2	R/W	0	0x020
MR3	Match Register 3	R/W	0	0x024

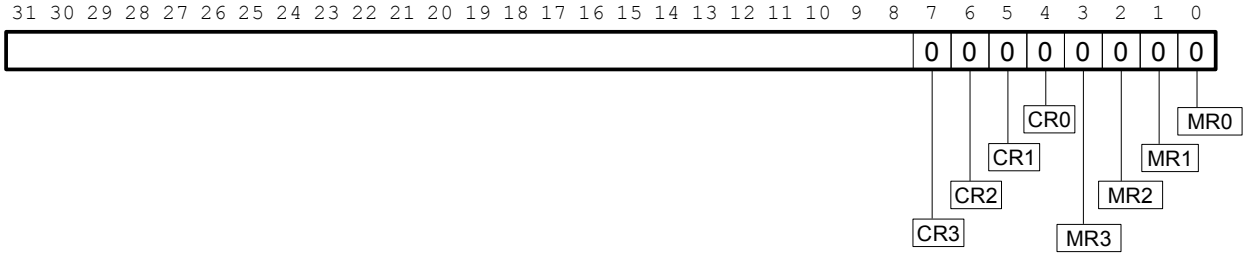
CCR	Capture Control Register	R/W	O	0x028
CR0	Capture Register 0	RO	0	0x02C
CR1	Capture Register 1	RO	0	0x030
CR2	Capture Register 2	RO	0	0x034
CR3	Capture Register 3	RO	0	0x038
EMR	External Match Register	R/W	0	0x03C
CTCR	Count Control Register	R/W	0	0x070

Partilha de pinos

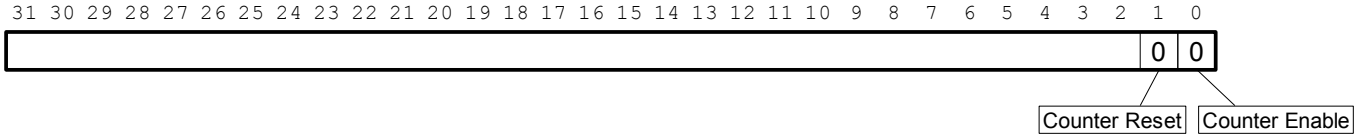
CAP0.0	P0.2	I
CAP0,1	P0.4	I
CAP0.2	P0.6	I
CAP1.0	P0.10	I
CAP1.1	P0.11	I
CAP1.2	P0.17	I
CAP1.3	P0.18	I

MAT0.0	P0.3	O
MAT0,1	P0.5	O
MAT0.2	P0.16	O
MAT1.0	P0.12	O
MAT1.1	P0.13	O
MAT1.2	P0.19	O
MAT1.3	P0.20	O

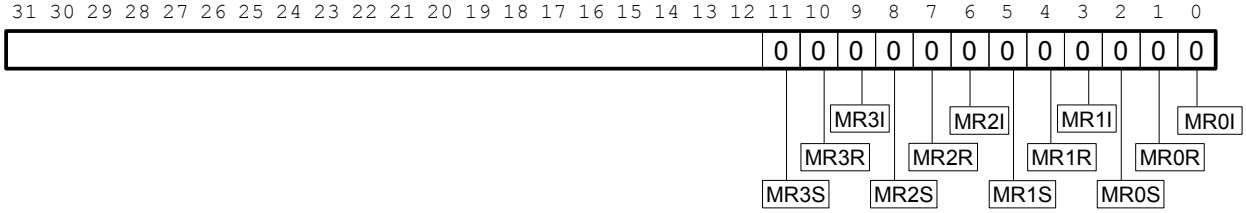
Interrupt Register



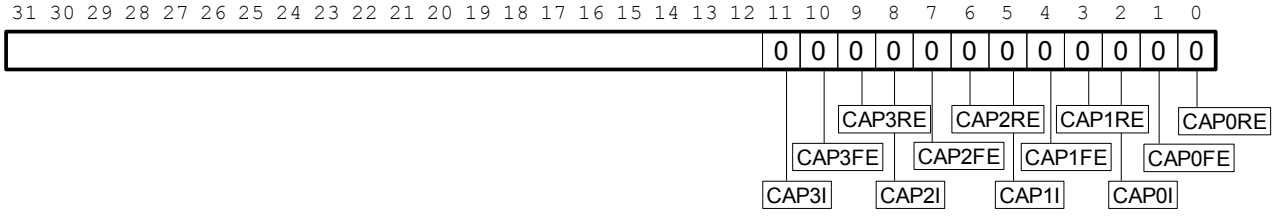
Timer Control Register



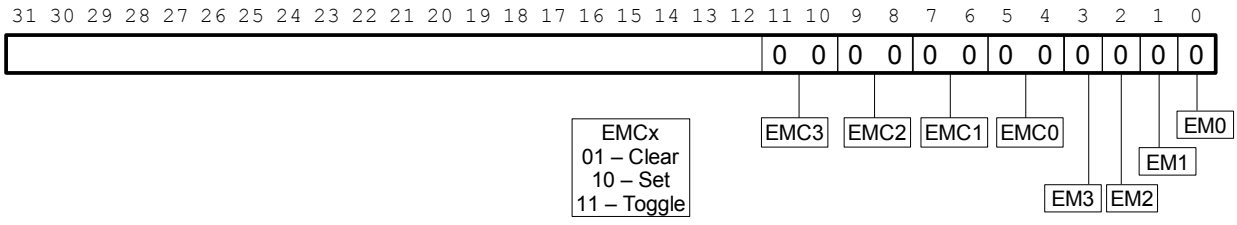
Match Control Register



Capture Control Register



External Match Register



Exemplo de programação

Se PCLK for igual a 14547600 Hz, para colocar Timer Counter a incrementar ao ritmo de 1 ms o PR deve ser programado com 14547.

O seguinte conjunto de funções permite efectuar medidas de tempo. A função timer_init coloca o contador do timer a contar ao ritmo de 1 Khz. A função timer_start regista o momento inicial. A função timer_elapsed retorna o tempo decorrido desde o momento inicial até agora.

```
void timer_init() {
    io_write_u32(LPC210X_T0 + LPC2XXX_TxPR, PCLK / 1000);
    io_write_u32(LPC210X_T0 + LPC2XXX_TxTCR, LPC2XXX_TxTCR_CTR_RESET);
    io_write_u32(LPC210X_T0 + LPC2XXX_TxTCR, LPC2XXX_TxTCR_CTR_ENABLE);
}

U32 timer_start() {
    U16 initial;
    io_read_u32(LPC210X_T0 + LPC2XXX_TxTC, initia);
    return initial;
}

U32 timer_elapsed(U32 initial) {
    U16 now;
    io_read_u32(LPC210X_T0 + LPC2XXX_TxTC, now);
    return now - initial;
}
```

Modo Capture

Cada registo de captura está associado a uma entrada CAP. Captura o valor corrente de TC quando ocorre uma mudança de estado desse pino.

Medir uma frequência baixa

O sinal é aplicado a CAP0. O valor de TC é capturado numa transição de CAP0. A diferença entre duas leituras consecutivas é o período do sinal a medir.

O limite de utilização deste método depende da latência do processador, isto é, o tempo que medeia entre uma transição do sinal e a leitura do registo *capture* por parte do processador.

Modo Match

Os registos match MR0 a MR3 são constantemente comparados com o TC. Quando um registo *match* é igual ao TC pode-se:

- Parar o TC e o PR;
- Colocar o TC a zero;
- Actuar uma saída externa MAT0 a MAT3 para zero, para um ou inverter o estado;
- Gerar uma interrupção.

Gerar uma onda quadrada com 50% de duty cycle.

Carregar um registo *match* com o número de ciclos de TC correspondente a metade do período. Quando o TC for igual ao registo *match,* fazer *reset* ao TC e inverter a respectiva saída MAT (um registo *match* está associada a uma saída MAT específica).

Frequência desejada: 2 MHz.

Com PR igual a zero a frequência aplicada ao TC é igual PCLK.

Assumindo PCLK = CCLK = 14457600 * 4 = 57830400

MR0 = 57830400 / 2000000 / 2 / 2 = 7

MCR, MR0R = 1 reset ao TC quando TC for igual a 7.

EMR, EM0 = 1 – selecciona saída MAT0

EMR, EMC0 = 2 – inverte a saída MAT0 a cada *match*.

(Como gerar um sinal periódico com duty cycle diferente de 50 %?)

Modo counter

Em modo *counter*, o TC em vez de incrementar com a saída de PC, incrementa com os impulsos externos aplicados a uma entrada CAP0 a CAP3. No registo CTCR define-se a transição que faz incrementar o contador e selecciona-se a entrada. A entrada seleccionada é sincronizada com PCLK. Por isso a duração de um patamar do sinal de entrada a zero ou a um não deve ser inferior a 1/PCLK.

Este modo pode ser usado para medir frequências altas até metade de PCLK. Fazendo leituras de TC no inicio e no fim de um intervalo de tempo definido obtém-se directamente o valor da frequência.

Pulse Width Modulator (PWM)

Esta unidade é baseada num *timer* comum. Não dispõe de entradas CAP e tem sete saídas – PWM0 a PWM6. As saídas são função dos registos MAT0 a MAT6.

Tem dois modos de funcionamento: controlo de transição simples e controlo de transição dupla.

Em ambos os casos o registo M0 controla a duração do ciclo.

No modo de transição simples todos as saídas são sincronizados no início do ciclo, durante o ciclo mudam de estado quando o respectivo registo match fica igual ao contador.

No modo de transição dupla são usados dois registos por cada saída. Um controla a primeira transição e o outro controla a segunda transição. Neste modo só é possível usam-se três saídas.

Real Time Clock (RTC)

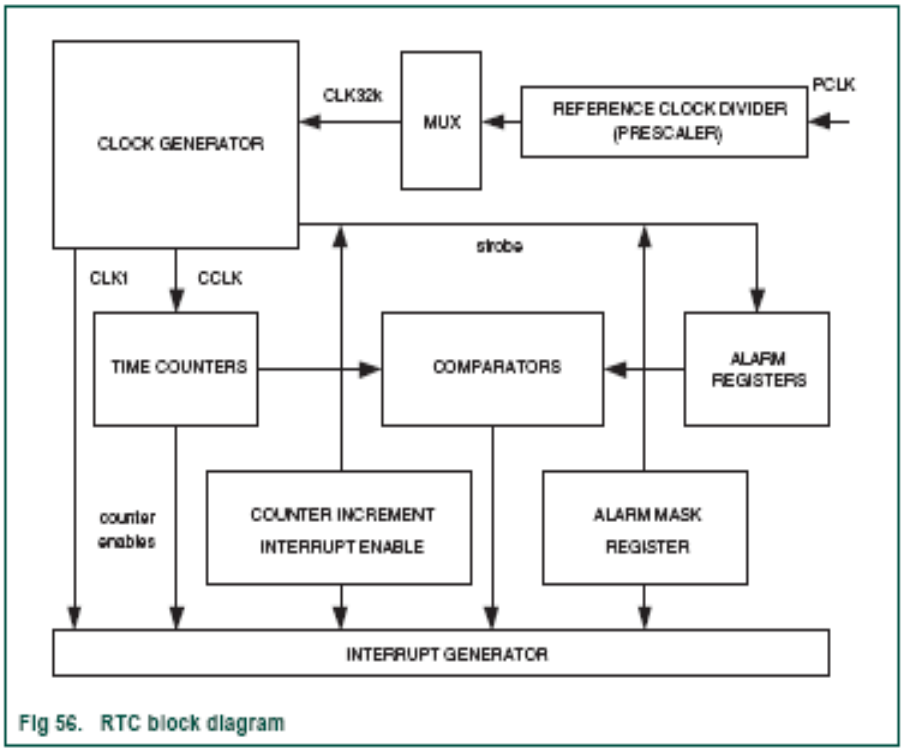


Fig 56. RTC block diagram

O relógio de tempo real (RTC) é baseado numa sequência de contadores encadeados.

Interface de programação

ILR	Interrupt Location	R/W	2	0x000
CTC	Clock Tic Counter	RO	15	0x004
CCR	Clock Control	R/W	4	0x008
CIIR	Counter Increment Interrupt	R/W	8	0x00C
AMR	Alarm Mask	R/W	8	0x010
CTIME0	Consolidated Time 0	RO	32	0x014
CTIME1	Consolidated Time 1	RO	32	0x018
CTIME2	Consolidated Time 2	RO	32	0x01C
SEC	Segundos	R/W	6	0x020
MIN	Minutos	R/W	6	0x024
HOUR	Horas	R/W	5	0x028
DOM	Dias do mês	R/W	5	0x02C
DOW	Dia da semana	R/W	3	0x030
DOY	Dia do ano	R/W	9	0x034
MONTH	Meses	R/W	4	0x038

ALSEC	Alarme - segundos	R/W	6	0x060
ALMIN	Alarme - minutos	R/W	6	0x064
ALHOUR	Alarme - horas	R/W	5	0x068
ALDOM	Alarme - dias do mês	R/W	5	0x06C
ALDOW	Alarme - dia da semana	R/W	3	0x070
ALDOY	Alarme - dia do ano	R/W	9	0x074
ALMON	Alarme - meses	R/W	4	0x078
ALYEAR	Alarme - anos	R/W	12	0x07C
PREINT	Prescaler integer portion	R/W	13	0x080
PREFRAC	Prescaler decimal portion	R/W	15	0x084

YEAR	Anos	R/W	12	0x03C					

ILR

31..2	1	0
	RTCCIF	RTCALF
	Counter increment interrupt	Alarm interrupt.

CCR

31..4	3..2	1	0
	CTTEST	CTCRST	CLKEN
	Test enable	CTC reset	Clock Enable

CIIR

31..8	7	6	5	4	3	2	1	0
	IMYEAR	IMMON	IMDOY	IMDOW	IMDOM	IMHOUR	IMMIN	IMSEC

AMR

31..8	7	6	5	4	3	2	1	0
	AMRYEAR	AMRMON	AMRDOY	AMRDOW	AMRDOM	AMRHOUR	AMRMIN	AMRSEC

CTIME0

31..27	26..24	23..21	20..16	15..14	13..8	7..6	5..0
	Day of week		Hours		Minutes		Seconds

CTIME1

31..28	27..16	15..12	11..8	7..5	4..0
	Year		Month		Day of month

CTIME2

31..12	11..0
	Day of year

Acertar o relógio

Para acertar o relógio deve-se para a contagem.

```
io_write_u32(LPC210X_RTC + LPC2XXX_RTC_CCR, LPC2XXX_RTC_CCR_CTCRST);
io_write_u32(LPC210X_RTC + LPC2XXX_RTC_SEC, tp->tm_sec);
io_write_u32(LPC210X_RTC + LPC2XXX_RTC_MIN, tp->tm_min);
io_write_u32(LPC210X_RTC + LPC2XXX_RTC_HOUR, tp->tm_hour);
io_write_u32(LPC210X_RTC + LPC2XXX_RTC_DOM, tp->tm_mday);
io_write_u32(LPC210X_RTC + LPC2XXX_RTC_DOW, tp->tm_wday);
io_write_u32(LPC210X_RTC + LPC2XXX_RTC_DOY, tp->tm_yday);
io_write_u32(LPC210X_RTC + LPC2XXX_RTC_MONTH, tp->tm_mon + 1);
io_write_u32(LPC210X_RTC + LPC2XXX_RTC_YEAR, 1900 + tp->tm_year);
io_write_u32(LPC210X_RTC + LPC2XXX_RTC_CCR, LPC2XXX_RTC_CCR_CLKEN);
```

Ler o relógio

A leitura pode ser feita nos registos anteriores ou, de uma forma compactada, nos registos CTIME0 a CTIME2. Como a leitura do relógio é composta pela leitura de vários registos em separado é necessário tomar precaução quanto à consistência do conjunto de valores lido. Se ocorrer um incremento dos contadores do RTC depois da leitura do primeiro registo e antes da leitura do último registo o conjunto data-hora obtido está errado.

```
aux0 = io_read_u32(LPC210X_RTC + LPC2XXX_RTC_CTIME0);
aux1 = io_read_u32(LPC210X_RTC + LPC2XXX_RTC_CTIME1);
aux2 = io_read_u32(LPC210X_RTC + LPC2XXX_RTC_CTIME2);
do {
     aux0a = aux0; aux1a = aux1; aux2a = aux2;
     aux0 = io_read_u32(LPC210X_RTC + LPC2XXX_RTC_CTIME0);
     aux1 = io_read_u32(LPC210X_RTC + LPC2XXX_RTC_CTIME1);
     aux2 = io_read_u32(LPC210X_RTC + LPC2XXX_RTC_CTIME2);
} while (aux0 != aux0a || aux1 != aux1a || aux2 != aux2a);
tp->tm_wday = (aux0 >> 24) & 7;
tp->tm_hour = (aux0 >> 16) & 0x1f;
tp->tm_min = (aux0 >> 8) & 0x3f;
tp->tm_sec = (aux0 & 0x3f);
tp->tm_year = ((aux1 >> 16) & 0xfff) - 1900;
tp->tm_mon = ((aux1 >> 8) & 0xf) - 1;
tp->tm_mday = aux1 & 0x1f;
tp->tm_yday = aux2 & 0xfff;
```

Frequência de referência

A montante da sequência de contadores do RTC há um contador de 15 bits – CTC que divide o relógio de referência por 32768 apresentando à saída um relógio de 1 Hz. O relógio de referência é gerado pelo bloco *prescaler*, com a frequência de 32768 Hz. Este valor é obtido a partir de PCLK por um processo de divisão fraccionária.

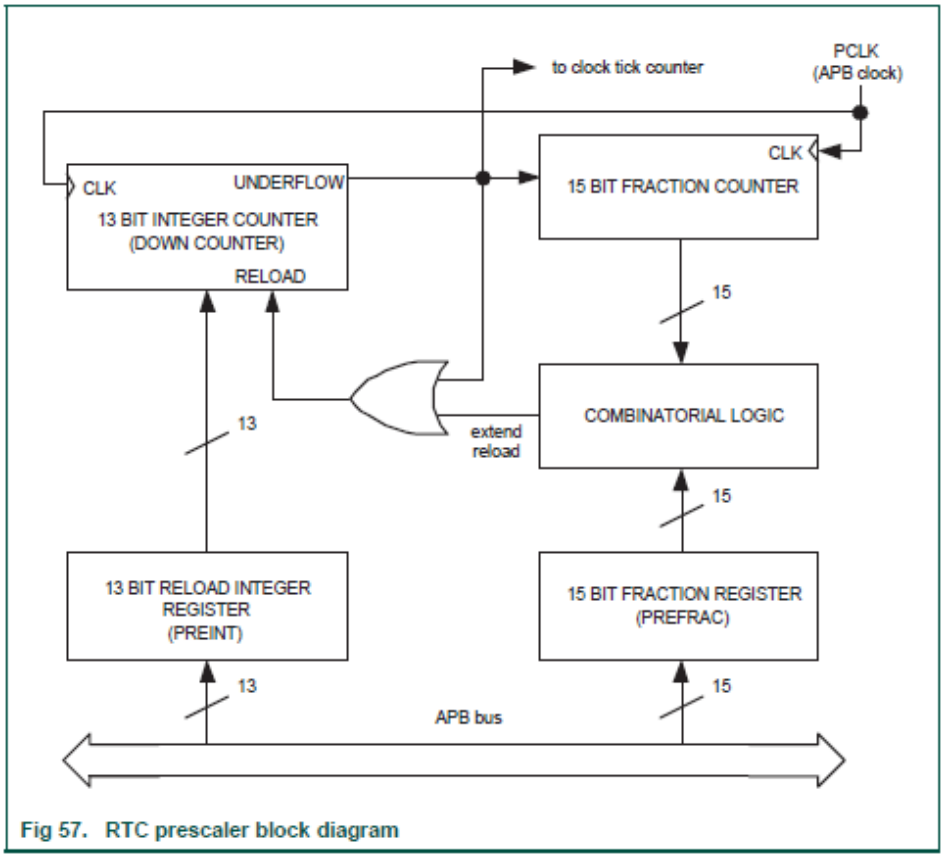


Fig 57. RTC prescaler block diagram

O bloco *prescaler* tem dois contadores e dois registos, um contador e um registo relacionado com a parte inteira da divisão (PREINT counter e PREINT register) e um contador e um registo relacionado com a parte fraccionária da divisão (PREFRAC counter e PREFRAC register).

A divisão inteira ocorre em PREINT counter que conta em contagem decrescente e gera um impulso quando chega a zero. Nessa altura é recarregado com o PREINT register recomeçando um novo ciclo.

A divisão fraccionária consiste em aumentar alguns ciclos de contagem do PREINT counter em mais uma unidade.

Esse aumento é provocado por um recarregamento imediato ao recarregamento de fim de contagem. A decisão sobre esse recarregamento é baseada em PREFRAC counter e em PREFRAC register. Estes têm uma dimensão de 15 bits, necessariamente igual a CTC. PREFRAC counter é incrementado no fim das contagens de PREINT counter. Se PREFRAC register tiver o bit 14 a um, então metade das contagens deve ser prolongada e essa indicação pode ser dada pelo bit de menor peso de PREFRAC counter. Se PREFRAC register tiver o bit de peso 13 a um, então um quarto das contagens deve ser prolongado e essa indicação pode ser dada pela combinação 10 nos bits 1 e 0 de PREFRAC counter.

Os valores a carregar em PREINT *register* e PREFRAC *register* para se obter a frequência de referência, de 32768 Hz, a partir de PCLK, são calculados da seguinte forma:

```
PREINT = (PCLK / 32768) – 1
```

```
PREFRAC = PCLK – (PREINT + 1) x 32768
```

Para o caso da frequência de PCLK ser 14745600

PREINT = (14745600 / 32768) – 1 = 449

PREFRAC = PCLK – (PREINT + 1) x 32768 = 0

```
io_write_u32(LPC210X_RTC + LPC2XXX_RTC_PREINT, (PCLK / 32768) – 1);
io_write_u32(LPC210X_RTC + LPC2XXX_RTC_PREFRAC, PCLK - (PCLK / 32768) * 32768);
```

Interrupções

Podem ser geradas interrupções quando qualquer contador muda de estado ou quando o RTC atinge os valores dos registos de alarme.

Memória Flash

A memória Flash existente no LPC2106 é do tipo NOR. Estas memórias possuem as seguintes características:

Leitura de dados

- O ciclo de escrita é igual ao da memória estática.
- As posições de memória podem ser lidas aleatoriamente.

Escrita de dados

- A escrita é uma operação demorada. A sua consumação demora mais que um ciclo de acesso.
- As operações de escrita apenas mudam uns para zeros. Para mudar de zeros para uns é necessário executar a operação de apagamento. A unidade mínima de apagamento é o sector.
- A operação de apagamento coloca todos os bits de um sector a um.
- Só se pode aceder a outras posições da mesma memória depois de terminar a operação corrente, mesmo que seja para leitura.

Flash do LPC2106

A Flash existente no LPC2106 tem 16 sectores de 8 Kbyte, numerados de 0 a 15. Os seus endereços vão de 0x00000000 a 0x0001FFFF. O últimos sector é ocupado pelo Boot Block e não pode ser alterado.

Operações disponíveis

- **Apagar sectores**
- **Escrever bloco de dados**. A dimensão do bloco de dados a escrever e o endereço onde se vai escrever devem ser um múltiplo de 512 bytes.
- **Preparar operação de escrita ou apagamento**. Para evitar a corrupção acidental dos dados gravados, as operações de escrita ou apagamento estão normalmente bloqueadas. Para as desbloquear é necessário executar esta operação antes de uma escrita ou apagamento. Após a operação de escrita os sectores voltam a ficar automaticamente protegidos.

Estas operações são realizadas por software gravado no Boot Block que é invocado pela aplicação através da interface IAP.

Esta interface define a existência de uma rotina de atendimento de pedidos de operações no endereço **0x7FFFFFF0** compatível com o seguinte protótipo:

```
void iap_entry(int iap_command[], int iap_result[]).
```

O primeiro parâmetro, passado em **r0,** é um *array* de inteiros onde se codificam os comando e os respectivos parâmetros. O segundo parâmetros, passado em **r1**, é o endereço de um *array* onde se codificam as respostas.

O software do Boot Block é executado em modo Thumb, pelo que, encontrando-se a aplicação a executar em modo ARM, deve-se providenciar a necessária mudança de modo. Esta mudança é realizada pela instrução **bx**, se o endereço de salto tiver o bit de menor peso a um. O retorno faz-se com uma instrução semelhante, usando o conteúdo de **r14** como endereço de retorno.

```
iap_entry:
    ldr  r12, =0x7ffffff1
    bx   r12
```

Exemplo de operação de apagamento

Para apagar um único sector **sector_start** e **sector_end** são iguais.

```
iap_command[0] = 50;                /* Prepair command */
iap_command[1] = sector_start;
iap_command[2] = sector_end;
iap_entry(iap_command, iap_result);

iap_command[0] = 52;                /* Erase sector command */
iap_command[1] = sector_start;
iap_command[2] = sector_end;
iap_command[3] = CCLK / 1000;           /* CPU clock in KHz */
iap_entry(iap_command, iap_result);
```

Exemplo de operação de escrita

O número de sector é obtido dividindo o endereço de memória pela dimensão dos sectores.

```
iap_command[0] = 50;                    /* Prepair to write */
iap_command[1] = (int)address / SECTOR_SIZE; /* Start sector */
iap_command[2] = (int)(address + size - 1) / SECTOR_SIZE;  /* End sector */
iap_entry(iap_command, iap_result);

iap_command[0] = 51;            /* Write command */
iap_command[1] = (U32) address; /* Address in FLASH where to write to */
iap_command[2] = (U32) data;        /* Address in RAM where to read from */
iap_command[3] = size;          /* Block size, must be 512, 1024 or 4096 */
iap_command[4] = CCLK / 1000;       /* CPU clock in KHz */
iap_entry(iap_command, iap_result);
```

Universal Asynchronous Receiver/Transmitter (UART)

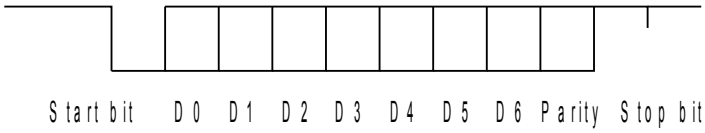
Conceitos básicos

Sincronização ao nível do bit – define os momentos em que o sinal recebido é amostrado para produzir a sequência de bits.

Sincronização ao nível do carácter – define a posição relativa dos bits.

Sincronização ao nível do bloco – define a posição de cada carácter.

Protocolo start-stop

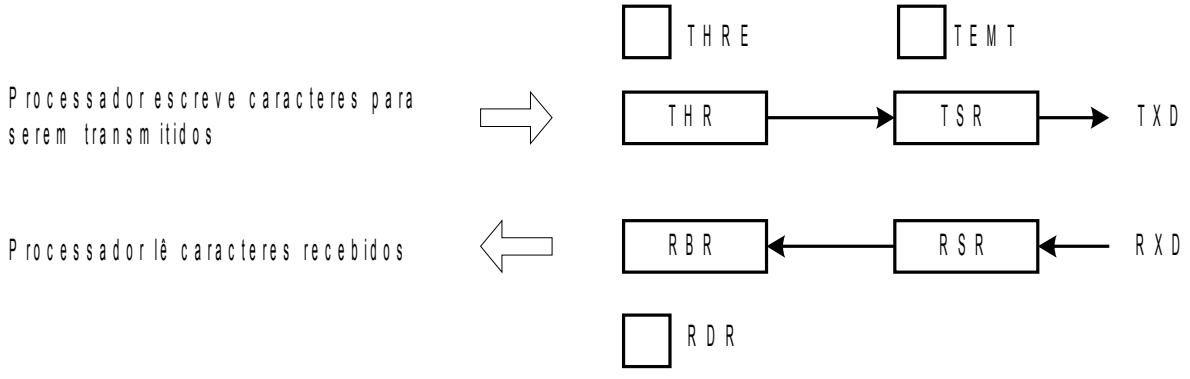


- A sincronização ao nível do bit pode ser feita por clock ou acertando a fase de um oscilador local com o flanco descendente do start bit.
- A sincronização ao nível do carácter é feita pelo start bit.
- O número de bits de dados pode ir de 5 a 8.
- A paridade pode não existir, ser par ou impar.
- O stop bit pode ter a duração de 1, 1,5 ou 2 bit time.
- Este protocolo não define sincronização ao nível do bloco.

Outros protocolos (HDLC)

- A sincronização ao nível do bit é feita por sinal de clock ou pela utilização de um código auto-sincronizável. O bit-stuffing e o código NRZI são muito usados.
- A sincronização ao nível do carácter e do bloco baseia-se na utilização de caracteres especiais (caracteres flag).

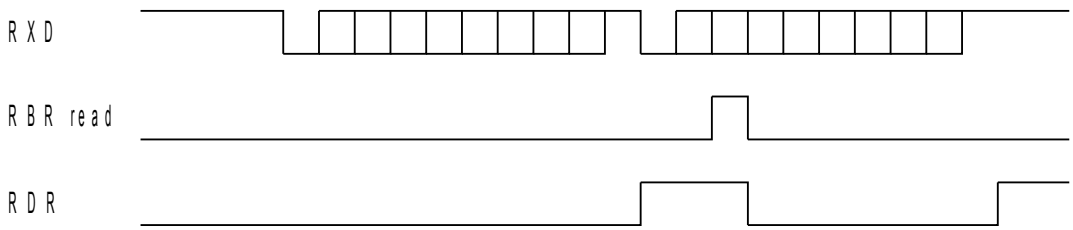
Um UART é composto por uma unidade de transmissão e uma unidade de recepção independentes. Ambas são compostas por um par de registos: um registo deslizante para transmissão ou recepção dos bits e outro para guarda dos caracteres.



Recepção

Depois de terminada a recepção dos bits que formam uma *frame* de caracter, no registador deslizante RSR, o carácter resultante é transferido para o registo RBR para ser lido pelo processador.

Enquanto o caracter aguarda, no registo RBR, a leitura por parte do processador a flag RDR é activada para sinalizar essa situação. Quando o processador recolhe o caracter a flag é desactivada.



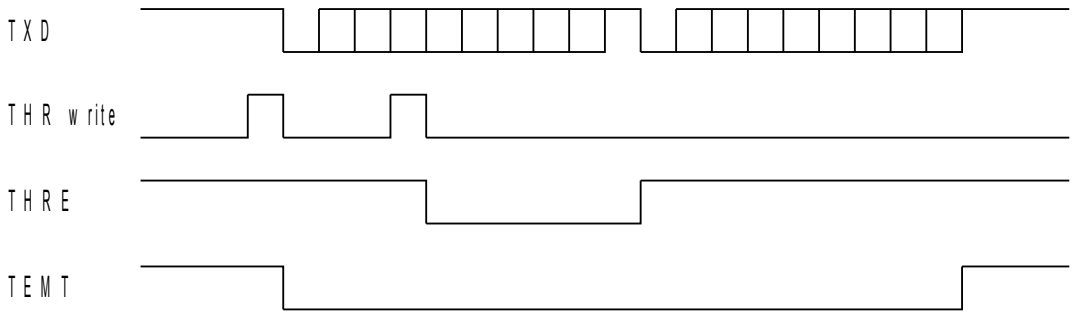
A função de recepção de caracteres (**uart_read_char**), só recolhe o novo caracter do registo RBR, depois de verificar, através da *flag* RDR, que existe novo caracter disponível.

```
U8 uart_read_char() {
    U32 aux;
    do {
         io_read_u32(uart_base_address + LPC2XXX_UxLSR, aux);
    } while ((aux & LPC2XXX_UxLSR_RDR) == 0);
    io_read_u32(uart_base_address + LPC2XXX_UxRBR, aux);
    return aux;
}
```

Transmissão

Na transmissão, os caracteres são escritos pelo processador no registo de transmissão THR. Em seguida são transferidos para o registo deslizante TSR para transmissão. Enquanto se estiver a processar uma transmissão o próximo caracter a transmitir aguarda em THR. No final de uma transmissão TSR fica livre. Se, nessa altura, existir caracter em THR este é automaticamente transferido para TSR iniciando-se imediatamente a nova transmissão.

As flasgs THRE e TEMT reflectem, respectivamente, o estado dos registos THR e TSR. Ambas são activadas quando o respectivo registo está vazio. Porque os caracteres são automaticamente transferidos de THR para TSR, pode concluir-se que: se THRE estiver desactivada ambos os registos estão ocupados e se TEMT estiver activada ambos os registos estão livres.



A função de envio de caracteres (**uart_write_char**), só deposita o novo caracter do registo RHR,

depois de verificar, através da *flag* THRE, que esse registo está livre.

```
void uart_write_char(U8 c)  {
    U32 aux;
    do {
         io_read_u32(uart_base_address + LPC2XXX_UxLSR, aux);
    } while ((aux & LPC2XXX_UxLSR_THRE) == 0);
    io_write_u32(uart_base_address + LPC2XXX_UxTHR, c);
}
```

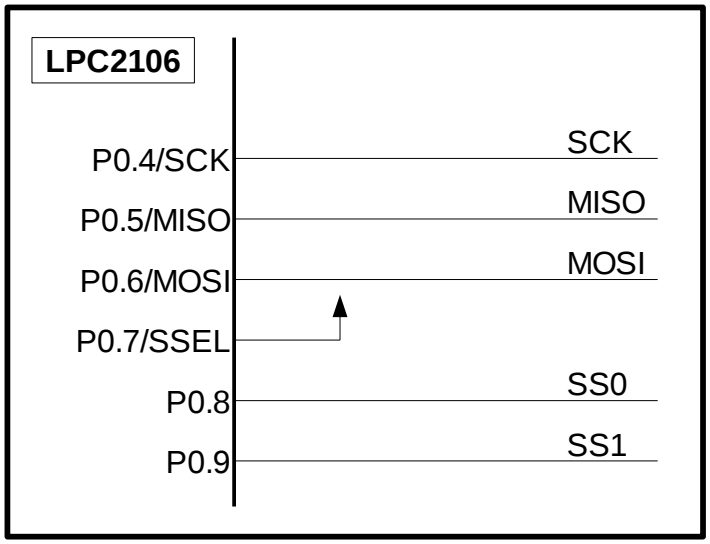
A flag TEMT serve para indicar à aplicação quando é que uma a transmissão foi efectivamente concluída. Este conhecimento pode ser necessário para que ao desligar a energia do sistema isso não suceda antes de terminarem as comunicações.

Ritmo da comunicação

```
U32 br_factor = input_clock / (baud_rate << 4);
io_write_u32(uart_base_address + LPC2XXX_UxLCR, LPC2XXX_UxLCR_DLAB);
io_write_u32(uart_base_address + LPC2XXX_UxDLL, br_factor);
io_write_u32(uart_base_address + LPC2XXX_UxDLM, br_factor >> 8);
```

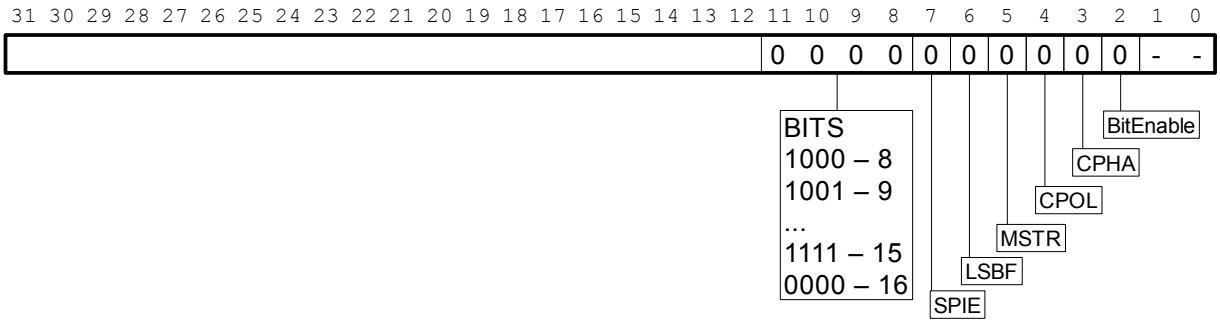
SPI

O LPC2106 tem integrado um controlador SPI, que pode operar em modo *master* ou em modo *slave*. Em modo *slave* o sinal SSEL funciona como entrada de selecção. Em modo *master* deve ser ligado a um (depende da versão).

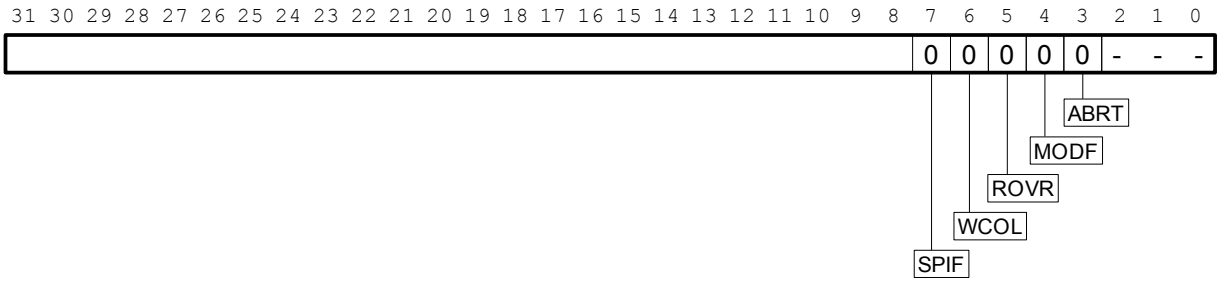


A interface de programação é composta por um registo de controlo onde se programam os parâmetros do protocolo – SPCR, um registo de estado – SPSR, um registo de dados – SPDR, um registo de definição da velocidade de transmissão – SPCCR e um registo de controlo de interrupções – SPINT.

SPI Control Register, 0xE0020000, SPCR, RW



SPI Status Register, 0xE0020004, SPSR, RO



Em seguida apresenta-se um exemplo de iniciação da interface SPI.

Começa-se por programar o relógio de trabalho na unidade de controlo da energia, em seguida programam-se os pinos na unidade de controlo dos pinos.

```
aux = io_read_u32(LPC210X_SCB + LPC2XXX_PCONP);
io_write_u32(LPC210X_SCB + LPC2XXX_PCONP, aux | LPC2XXX_PCONP_SPI);

io_write_u32(LPC210X_SPI + LPC2XXX_SPCR, LPC2XXX_SPCR_MSTR);

aux = io_read_u32(LPC210X_PCB + LPC2XXX_PINSEL0)
    & LPC2XXX_PINSEL0_GPIO(SCK_PIN) & LPC2XXX_PINSEL0_GPIO(MOSI_PIN) &
    LPC2XXX_PINSEL0_GPIO(MISO_PIN);
io_write_u32(LPC210X_PCB + LPC2XXX_PINSEL0, aux | LPC2XXX_PINSEL_SCK |
    LPC2XXX_PINSEL_MOSI | LPC2XXX_PINSEL_MISO | LPC2XXX_PINSEL_SSEL);
```

Os parâmetros de comunicação dependem do periférico, por isso a operação de programação dos parâmetros da comunicação deve ser executada sempre que se muda de periférico.

A velocidade de comnicação é igual a PCLK a dividir pelo valor do registo SPCCR.

Os parâmetros de comunicação passíveis de serem programados são:

- o número de bits da palavra (apenas no LPC2106, versão 01);
- a polaridade e fase do sinal de relógio;
- o modo de operação do controlador – *master* ou *slave*;
- a ordem de transmissão dos bits – LSB ou MSB.

```
    io_write_u32(LPC210X_SPI + LPC2XXX_SPCCR, spi->clock);
    io_write_u32(LPC210X_SPI + LPC2XXX_SPCR, LPC2XXX_SPCR_MSTR |
        ((spi->mode & 2) << (LPC2XXX_SPCR_CPOL_SHIFT - 1)) |
        ((spi->mode & 1) << LPC2XXX_SPCR_CPHA_SHIFT));
```

A transferência de uma palavra processa-se do seguinte modo:

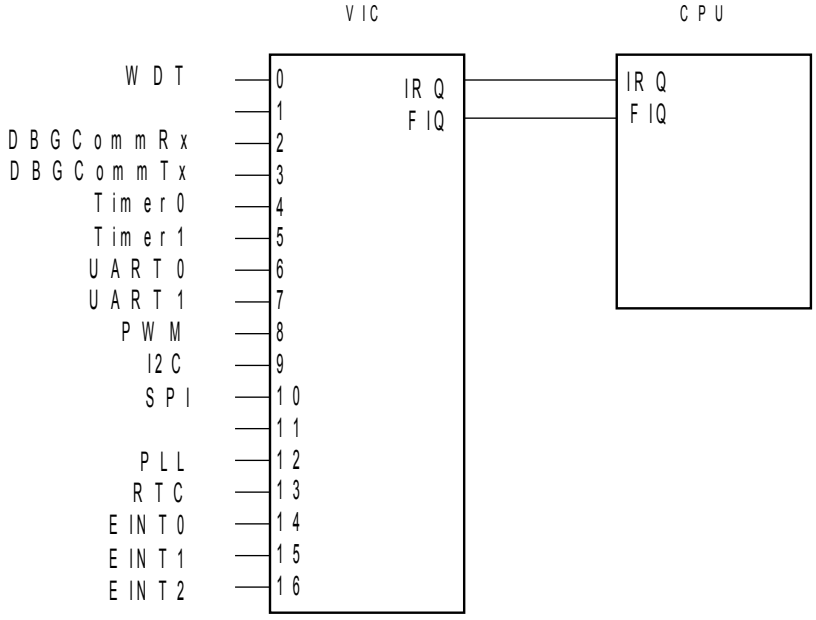
- escreve-se a palavra a transmitir no registo de dados;
- esta escrita desencadeia a transferência;
- em seguida aguarda-se o fim da transferência, pesquisando no registo de estado a *flag* SPIF;
- por fim recolhe-se a palavra recebida do registo de dados.

```
    io_write_u32(LPC210X_SPI + LPC2XXX_SPDR, *tx_data++);
    while ((io_read_u32(LPC210X_SPI + LPC2XXX_SPSR) & LPC2XXX_SPSR_SPIF) == 0)
        ;
    *rx_buffer++ = io_read_u32(LPC210X_SPI + LPC2XXX_SPDR);
```

Interrupções

O processador ARM possui duas entradas de interrupção: IRQ e FIQ. Associado ao processador existe um periférico controlador de interrupções que recebe os pedidos de interrupção dos periféricos e os encaminha para o processador através de IRQ e FIQ

.



Funcionamento do controlador de interrupções

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

As fontes associadas a IRQ podem ser vectorizadas, isto é, ser indicado directamente 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 deve inquirir o controlador de interrupções no registo **VICIRQStatus** para determinar quais os periféricos que activaram o pedido e que devem ser servidos.

Um periférico ao ser servido retira o pedido de interrupção.

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

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.

As fontes que activam a entrada FIQ não são vectorizadas. No caso, pouco usual, 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 ficar activa. Esta funcionalidade permite que

rotinas associadas a uma menor prioridade sejam interrompidas para dar lugar ao atendimento de interrupções 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.

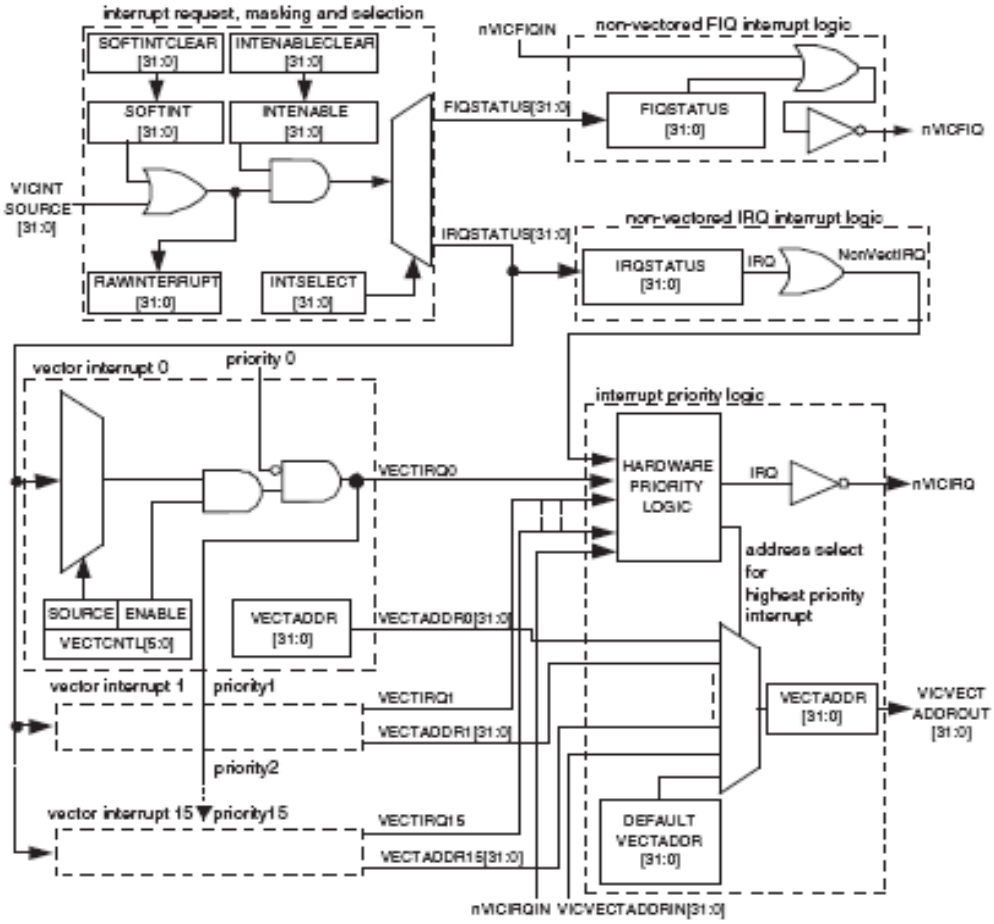


Fig 13. Block diagram of the Vectored Interrupt Controller

Interrupções externas

As fontes de interrupção são designadas por internas quando são geradas por periféricos internos ao microcontrolador ou são designadas por externas quando são geradas por periféricos externos.

Nos processadores da família LPC os sinais de interrupção externos são tratadas pela unidade System Control Block antes de serem aplicados ao controlador de interrupções – VIC. No VIC as fontes de interrupção internas e externas são tratadas de maneira uniforme.

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 existem na versão 01 do LPC2106. Nas outras versões, é como se os valores destes registos fossem zero. As entradas de interrupção externas são sensíveis ao nível zero, ou seja, é gerada interrupção se, e sempre que, uma destas entradas esteja a zero.

Funcionamento do processador

Como resposta à activação dos sinais IRQ ou FIQ, 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. Nestas posições de memória estão instruções que carregam no PC o endereço da rotina de serviço à interrupção.
No caso FIQ, o endereço da rotina de serviço **`fiq_isr`** encontra-se na posição de memória **`address_fiq_isr.`**

```
0x0000001C    ldr  pc, address_fiq_isr

     ...

0x00000030    address_fiq_isr:  .word      fiq_isr
```

No caso IRQ, o código normalmente usado é:

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

Esta instrução carrega no PC o conteúdo do registo **VICVectAddr**. O endereço deste registo é **`0xFFFFF030`** que é obtido subtraindo **`0xFF0`** a **`0x00000020.`** **`0x00000020`** é o valor de PC quando a instrução **`ldr pc, [pc, #-0xFF0]`** é executada.

O atendimento de uma interrupção implica a mudança de modo do CPU para modo IRQ ou 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 surge um o registo SPSR. No modo FIQ há a acrescentar a substituição dos registos R8 a R12 por registos próprios deste modo.

O registo SPSR contém o estado do registo CPSR do programa interrompido o que permite o regresso ao modo anterior com as flags no mesmo estado em que se encontravam.

O registo LR contém o endereço de retorno ao programa interrompido. A necessidade de criação de um novo registo LR no modo IRQ deve-se ao facto de ser necessário guardar este endereço e não se poder utilizar o LR do programa interrompido pois poderia estar em uso. O endereço exacto onde o processador deve retomar o processamento à obtido subtraindo 4 ao valor de LR.

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

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. 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. Nesta altura o processador está em modo IRQ com a *flag* I a 1 - interrupções inibidas. O controlador de interrupções também assinalou a entrada numa rotina de interrupção de determinado nível de prioridade. Se a fonte de interrupção for vectorizada a rotina de interrupção 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. 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 usual). Nesta altura o controlador é também informado do fim do atendimento. Se existir outro pedido activo (ou permanecer o mesmo) o sinal IRQ é imediatamente 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 o programa interrompido e restabelecer a *flag* I a 0 (activada) inicia novamente um novo ciclo de atendimento.

Iniciação das interrupções

Como preparação para o atendimento de interrupções é necessário:

- Garantir que a tabela de excepções está bem formada

```
    .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
```

  e 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);
```

- 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;

```
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
```

No ensaio de um programa pode acontecer reiniciar-se o programa sucessivamente sem fazer reset ao hardware. O controlador de interrupções pode ter memorizado fffffffffffff

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);
```

No ensaio de um programa pode acontecer reiniciar-se o programa sucessivamente sem fazer reset ao hardware. 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);
```

Aninhamento de Interrupções

Interrupções falsas