REDS
Reconfigurable & embedded Digital Systems

heig-vd
HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD
www.heig-vd.ch

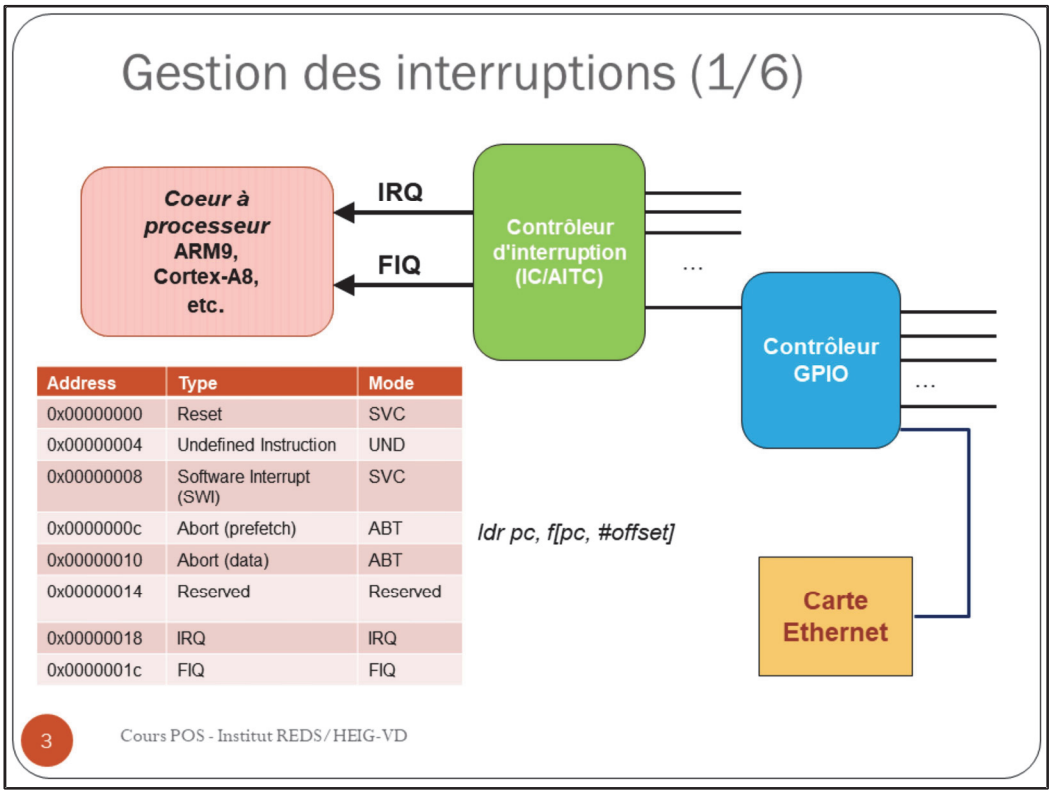
Portage de systèmes d'exploitation (POS)
Evénements asynchrones

Prof. Daniel Rossier

Version 1.3 (2017-2018)

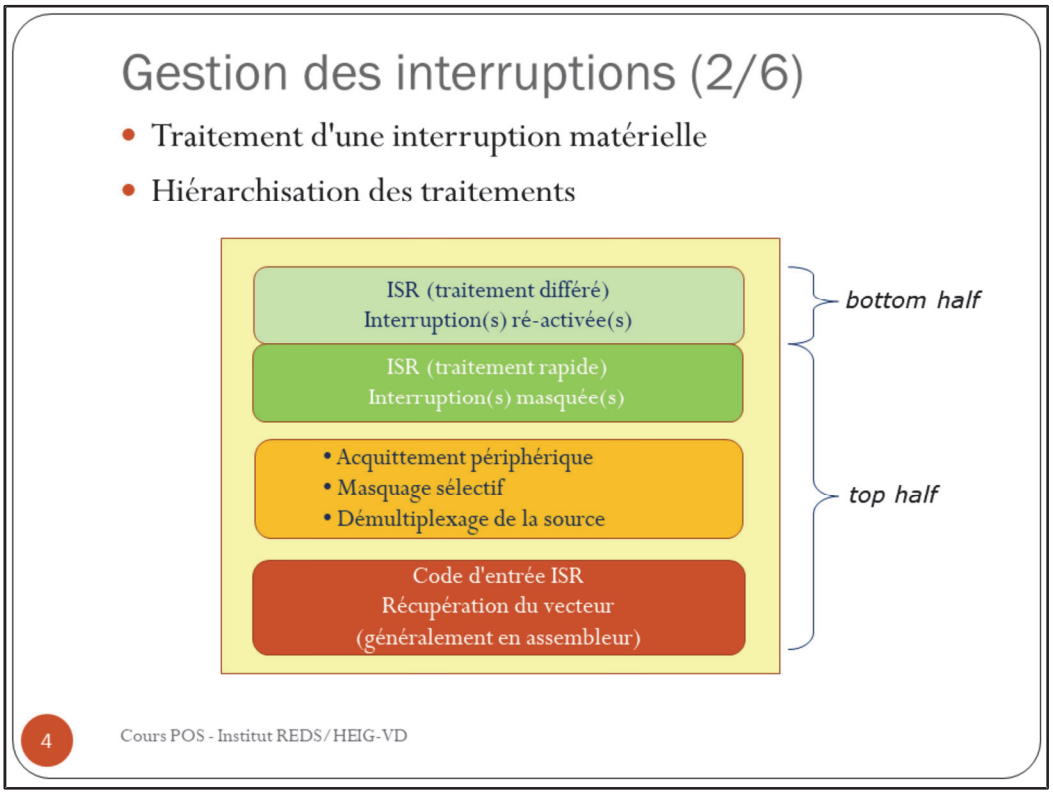
Plan

- Gestion des interruptions
- Gestion du temps



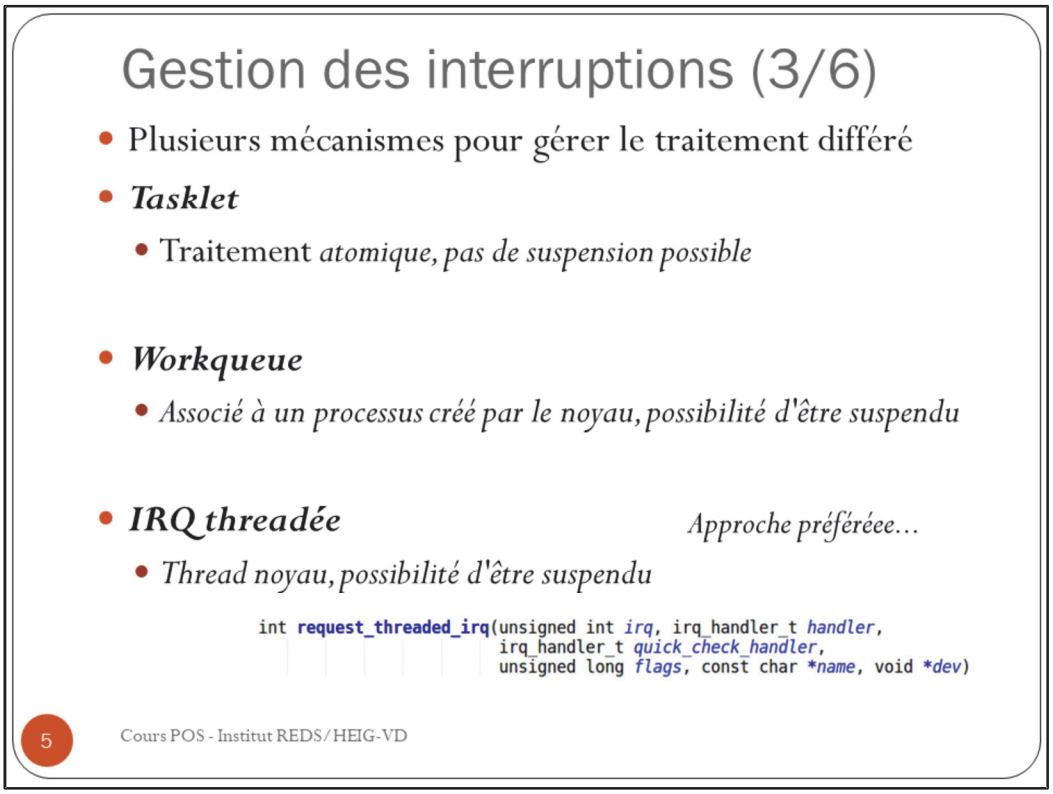
L'adresse de l'emplacement des vecteurs d'interruption peut varier selon le microcontrôleur. Le contenu de la table peut également changer dynamiquement pour être plus efficace dans l'appel des routines de service.

Selon le type de microcontrôleur, il est possible d'utiliser un coprocesseur afin de récupérer directement le numéro de la source sur le contrôleur d'interruption.



Dans la majorité des cas, le traitement lié à une interruption matérielle se décompose en plusieurs niveaux de fonction. Le premier niveau (*low-level handler*) concerne la gestion du mode d'exécution, l'acquittement de bas niveau (contrôleur d'interruption) si nécessaire, un premier démultiplexage de l'interruption. Par démultiplexage, on entend la conversion d'un vecteur d'entrée vers un autre vecteur amenant l'exécution d'une routine de niveau supérieur. Cette dernière effectuera un traitement plus évolué, poursuivra le traitement des acquittements, du masquage des interruptions, et du démultiplexage de la source (niveau GPIO par exemple). Finalement, un dernier niveau concerne le traitement spécifique qui est lié au périphérique à l'origine de l'interruption.

Cette chaîne de traitement s'effectue généralement dans le contexte de l'interruption et est considéré comme la partie de traitement rapide (*top half*). En effet, le code en cours d'exécution a été interrompu, et les sous-systèmes n'ont pas la main durant le traitement de l'interruption. C'est pourquoi, le code lié au traitement rapide doit être généralement court et donner la main à un "code de relais" qui constitue le traitement différé (*bottom half*). Ce code sera exécuté "sous contrôle" de l'ordonnanceur du noyau.

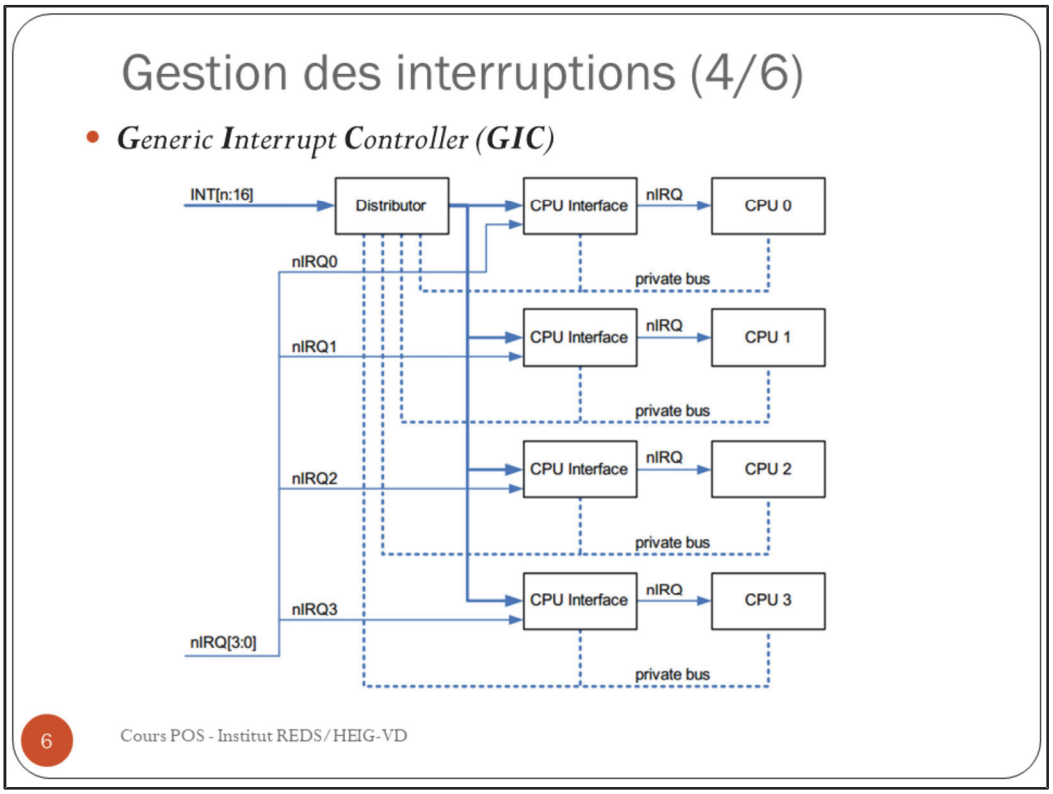


Dans *Linux,* il existe plusieurs mécanismes permettant l'implémentation de traitements différés. Les trois principaux sont les *tasklets*, les *workqueues* et les *IRQ threadées.*

Les ***tasklets*** sont exécutés de manière opportuniste typiquement à la fin de traitement d'une routine de service et d'une manière générale aux points de préemption dans le noyau. Le traitement ne peut se faire que de manière atomique et sérialisée, c-à-d que si plusieurs *tasklets* de même type doivent être exécutées (suite à plusieurs interruptions différentes par exemple), le traitement ne peut être parallélisé sur plusieurs CPUs.

Les ***workqueues*** associe le traitement à exécuter à un processus créé et géré par le noyau. Dans ce cas, le traitement pourrait endormir le *thread* courant dans le cas d'une attente sur une ressource ou d'un délai requis.

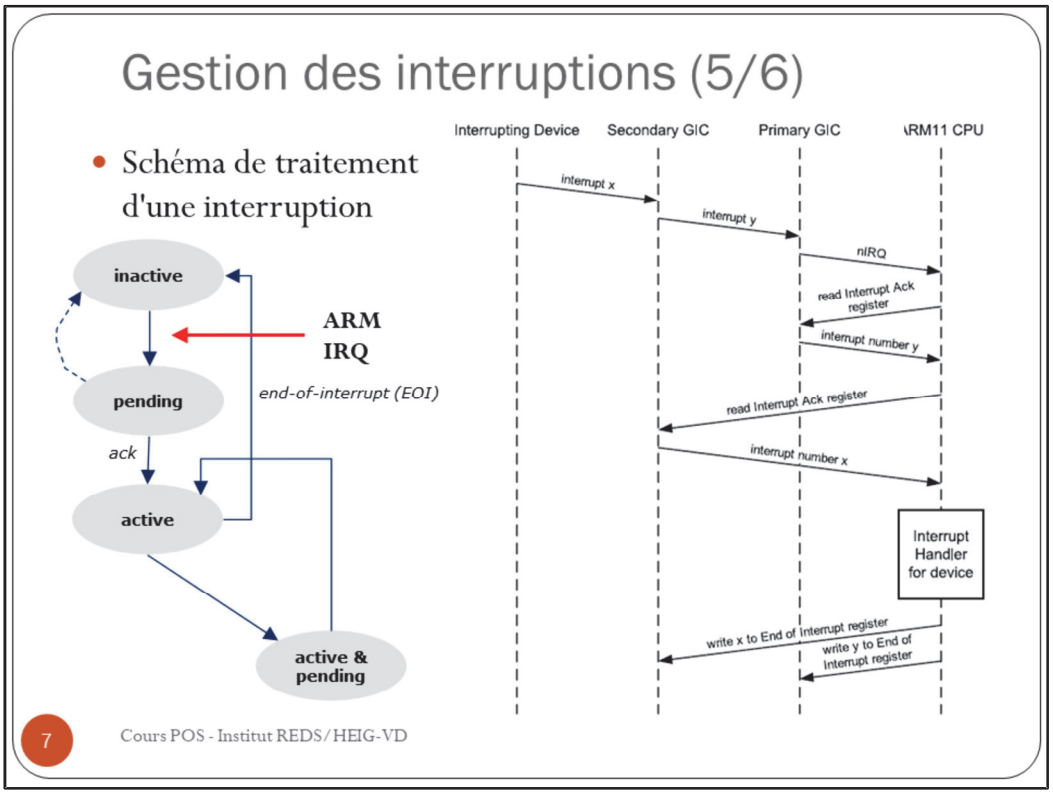
Les ***IRQs threadés*** sont proches de la notion de *workqueue* mais sont gérés entièrement par des *thread* du noyau. Il peuvent également se suspendre et possèdent une priorité plus élevée qu'un autre *thread*. Ils sont plus simples à gérer que les *workqueues* et peuvent être associés à un coeur CPU spécifique (affinité).
Il tendent à remplacer les *tasklets* dans le futur.



Les contrôleurs d'interruption dans les microcontrôleurs de la famille ARM tendent de plus en plus à être basés sur un même type de contrôleur appelé GIC (*Generic Interrupt Controller*).

L'architecture d'un GIC permet de gérer les interruptions sur un système multicoeur. Le GIC est décomposé en deux blocs fonctionnels principaux: le distributeur (*distributor*), qui constitue le contrôleur secondaire par lequel transite toutes les interruptions matérielles, et le contrôleur primaire appelé aussi *CPU Interface*. Ce dernier gère les interruptions au niveau d'un CPU. Ainsi, il y a autant de contrôleurs primaires que de CPUs.

Comme le montre la figure ci-dessus, l'architecture permet de considérer des lignes IRQs physiques reliées directement au contrôleur primaire afin de disposer d'un canal plus rapide.



Le flux de traitement lors d'une interruption gérée par un GIC est montré sur la figure ci-dessus avec le diagramme état-transition. A l'arrivée d'une interruption en provenance d'une source, le distributeur examine la configuration et la disponibilité des contrôleurs primaires actifs.

Puis l'interruption est transmise au contrôleur primaire qui prend en charge l'interruption. Cette dernière est transmise au CPU via le canal IRQ (ou FIQ) et l'état associé au numéro de la source passe alors de ***Inactive* à *Pending***. Le processeur lève l'interruption (IRQ) et la routine de service (ISR) associée est exécutée. L'ISR doit alors acquittée l'interruption qui conduira une transition de l'état ***Pending* à *Active***.
Lorsque l'ISR a terminé son travail, une requête de *End-Of-Interrupt* est alors envoyée au GIC qui passera l'état de la source de ***Active* à *Inactive*.**

Une interruption sur la même source peut intervenir lorsque l'état est dans *Active.* Dans ce cas, il n'y a pas d'interruption IRQ réémise (la ligne est désactivée en principe dans le traitement immédiat) et l'état de la source passe alors de ***Active* à *Active and pending***. C'est l'ISR de bas niveau qui sera responsable d'interroger à nouveau le contrôleur afin d'examiner la présence éventuelle d'interruptions pendantes.

Gestion des interruptions (6/6)

- Configuration de l'adresse de la table des vecteurs

```
int arch_irq_init (void) {

    __asm__(
        // Set vector table at address __vectors_start__
        "mcr p15, 0, %0, c12, c0, 0;":: "r" (__vectors_start__):
    );

    return 0;
}
```

- Descripteur du *irq_chip* associé au *GIC*

```
static struct irq_chip gic_chip = {
    .name       = "GIC",
    .irq_mask   = gic_mask_irq,
    .irq_unmask = gic_unmask_irq,
    .irq_eoi        = gic_eoi_irq,
    .irq_set_type   = gic_set_type,
    .irq_retrigger  = gic_retrigger,
#ifdef CONFIG_SMP
    .irq_set_affinity   = gic_set_affinity,
#endif
    .irq_set_wake   = gic_set_wake,
};
```

Gestion du temps (1/7)

- Types d'horloge
  - Horloge monotonique
    - Temps écoulé continu depuis le démarrage
    - ***clocksource***
  - Timer programmable (événementiel)
    - Génération des interruptions à une certaine fréquence
    - ***clockevent***
- Types de *timer*
  - *Timer* apériodique (*one shot*)
  - *Timer* périodique
    - *autoreload*

Gestion du temps (2/7)

- *jiffies*
  - Variable globale dans le noyau
  - Nombre de *tick timer* depuis l'activation du *timer*.
    - Un *jiffy* est lié à la notion de *tick* du *timer* système
  - Fréquence définie par la constante *HZ*
    - Valeurs typiques : *1000 (1 ms), 100 (10 ms), 250 (4 ms)*

```
unsigned long time_stamp = jiffies;           /* now */
unsigned long next_tick = jiffies + 1;        /* one tick from now */
unsigned long later = jiffies + 5*HZ;         /* five seconds from now */
```

  - Tend à être supprimé au profit d'un *timer* dynamique

Gestion du temps (3/7)

- Configuration d'une source monotonique

Extrait d'un code d'initialisation dans Linux 3.16 (arch/arm/common/timer-sp.c)

```
2  void __init __sp804_clocksource_and_sched_clock_init(void __iomem *base, const char *name,
3                                                       struct clk *clk, int use_sched_clock)
4  {
5      long rate;
6
7      if (!clk) {
8        clk = clk_get_sys("sp804", name);
9
10     /* ... */
11
12     /* setup timer 0 as free-running clocksource */
13     writel(0, base + TIMER_CTRL);
14     writel(0xffffffff, base + TIMER_LOAD);
15     writel(0xffffffff, base + TIMER_VALUE);
16     writel(TIMER_CTRL_32BIT | TIMER_CTRL_ENABLE | TIMER_CTRL_PERIODIC,
17         base + TIMER_CTRL);
18
19     clocksource_mmio_init(base + TIMER_VALUE, name, rate, 200, 32, clocksource_mmio_readl_down);
20
21     if (use_sched_clock) {
22         sched_clock_base = base;
23         sched_clock_register(sp804_read, 32, rate);
24     }
25 }
```

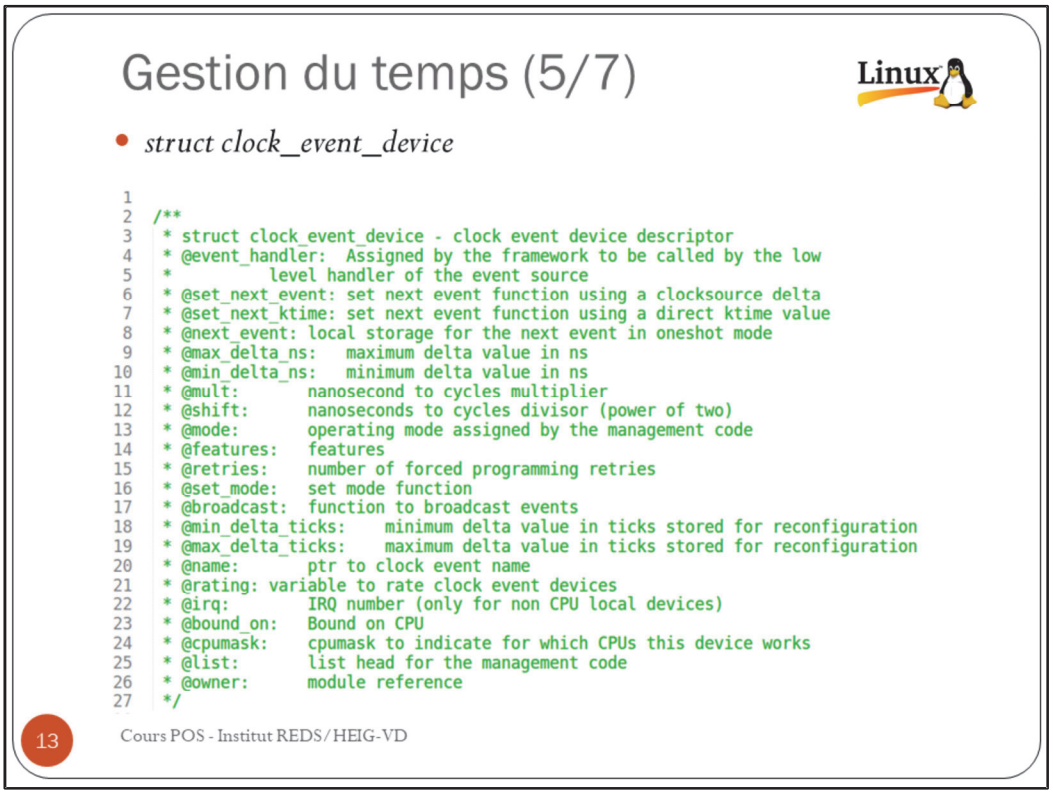
Gestion du temps (4/7)

- Configuration du *timer* événementiel

Extrait d'un code d'initialisation dans Linux 3.16 (arch/arm/common/timer-sp.c)

```
2  void __init __sp804_clockevents_init(void __iomem *base, unsigned int irq, struct clk *clk,
3                                       const char *name)
4  {
5      struct clock_event_device *evt = &sp804_clockevent;
6      long rate;
7
8      if (!clk)
9          clk = clk_get_sys("sp804", name);
10
11     /* ... */
12
13     rate = sp804_get_clock_rate(clk);
14     if (rate < 0)
15         return;
16
17     clkevt_base = base;
18     clkevt_reload = DIV_ROUND_CLOSEST(rate, HZ);
19     evt->name = name;
20     evt->irq = irq;
21     evt->cpumask = cpu_possible_mask;
22
23     writel(0, base + TIMER_CTRL);
24
25     setup_irq(irq, &sp804_timer_irq);
26     clockevents_config_and_register(evt, rate, 0xf, 0xffffffff);
27 }
```

```
2
3  static irqreturn_t sp804_timer_interrupt(int irq, void *dev_id)
4  {
5      struct clock_event_device *evt = dev_id;
6
7      /* clear the interrupt */
8      writel(1, clkevt_base + TIMER_INTCLR);
9
10     evt->event_handler(evt);
11
12     return IRQ_HANDLED;
13 }
14
15 static struct irqaction sp804_timer_irq = {
16     .name       = "timer",
17     .flags      = IRQF_TIMER | IRQF_IRQPOLL,
18     .handler    = sp804_timer_interrupt,
19     .dev_id     = &sp804_clockevent,
20 };
```



La conversion entre le nombre de cycles et une valeur de temps exprimée en ns peut s'effectuer avec la formule suivante :

```
ns = ((u64) cycles * mult) >> shift;
```

où *ns* représente le nombre de ns correspondant au nombre de *cycles.*

Les valeurs de conversions *mult* et *shift* sont calculées et permettent de fournissent une approximation de la conversion.

La conversion inverse peut s'effectuer avec la même formule.

Gestion du temps (6/7)

- *struct clock_source*

```
/**
 * struct clocksource - hardware abstraction for a free running counter
 *  Provides mostly state-free accessors to the underlying hardware.
 *  This is the structure used for system time.
 *
 * @name:       ptr to clocksource name
 * @list:       list head for registration
 * @rating: rating value for selection (higher is better)
 *          To avoid rating inflation the following
 *          list should give you a guide as to how
 *          to assign your clocksource a rating
 *          1-99: Unfit for real use
 *              Only available for bootup and testing purposes.
 *          ...
 *          400-499: Perfect :  The ideal clocksource. A must-use where available.
 * @read:       returns a cycle value, passes clocksource as argument
 * @enable: optional function to enable the clocksource
 * @disable:    optional function to disable the clocksource
 * @mask:       bitmask for two's complement subtraction of non 64 bit counters
 * @mult:       cycle to nanosecond multiplier
 * @shift:      cycle to nanosecond divisor (power of two)
 * @max_idle_ns:    max idle time permitted by the clocksource (nsecs)
 * @maxadj: maximum adjustment value to mult (~11%)
 * @flags:      flags describing special properties
 * @archdata:   arch-specific data
 * @suspend:    suspend function for the clocksource, if necessary
 * @resume: resume function for the clocksource, if necessary
 * @cycle_last: most recent cycle counter value seen by ::read()
 * @owner:      module reference, must be set by clocksource in modules
 */

```

Gestion du temps (7/7)

- Ajustage des délais

```
void calibrate_delay(void);
```

- Calcule le nombre d'itérations entre deux *jiffies*.
  - Combien d'instructions peuvent être exécutées entre deux interruptions du *timer* ?
- Permet d'implémenter une fonction d'attente avec de petits délais ***< 2 ms***
  - Attente active

Références

- Pierre Ficheux, *Linux embarqué*, 3e édition, Éditions Eyrolles (2010)
- Free Electrons. Kernel, drivers and embedded Linux development, consulting, training and support. http://free-electrons.com