Gestion des périphériques

MI074 - 7

Périphériques

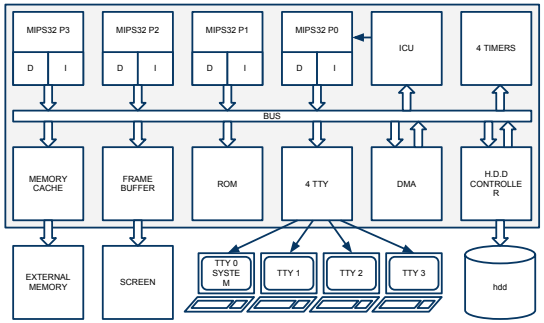
DMA : réalise des transferts de mémoire.
TTY : écrit des caractères sur l'écran et lit le clavier.
BD : réalise des transferts de bloc (p.ex 512o) entre le disque et la mémoire.
ICU : concentre les lignes d'interruptions des périphériques.
TIMER : produit des interruptions périodiques (cycles).
FB : tableau de pixels (2 octets / px) représentant une image.
CPU : ce n'est pas un périphérique par définition, mais il dispose d'une 'ICU' et pour réduire le nombre de type de structures C, il va être assimilé un périphérique dans certain cas.

Les périphériques / devices

- Un périphérique, a priori c'est tout ce qui n'est pas pas le processeur qui permet les échanges avec l'extérieur
  - terminal tty : cible + IT (interruption)
  - block device : cible et initiateur + IT
  - frame buffer : cible

- mais aussi des accélérateurs ou des composants de services
  - dma : cible et initiateur + IT
  - timer : cible + IT
  - icu : cible + IT

On utilise plutôt le terme **device** (dispositif matériel) plus général et donc plus adapté.



Abstraction de l'architecture

- On souhaites abstraire les périphériques,
  c'est à dire unifier leur gestion malgré leur diversité.

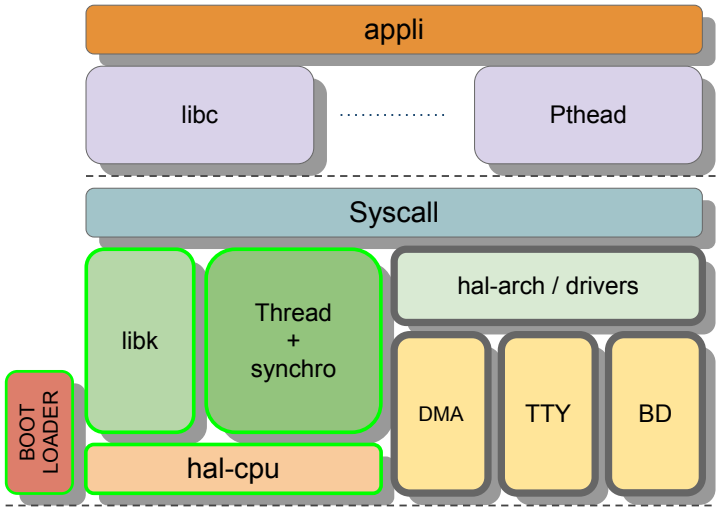
- On va définir un périphérique comme un objet générique
  qui va être spécialisé en fonction du périphérique réel.

- On va définir une collection de fonctions (une API)
  pour chaque type de périphérique dont:
  - Les prototypes sont imposés par le noyau.
  - Les implémentations sont spécifiques au type de périphérique.

- L'ensemble de ces fonctions constitue le pilote (driver)
  du périphérique (device)

- Cette API est normalement utilisée par des sous-systèmes de l'OS, le système de fichier ou la pile réseau que nous n'allons pas faire. Dans notre cas elle sera proposée à l'utilisateur par l'intermédiaire de syscall.

Architecture de l'OS final



Opérations sur les périphériques

DEV_ICU

```
void        set_mask        (device_t *dev, unsigned mask);
void        clr_mask        (device_t *dev, unsigned mask);
unsigned    get_mask        (device_t *dev);
unsigned    get_highest_irq (device_t *dev);
```

Les types
- device_t,
- dev_request_t et
- dev_param_t

sont décrits plus après

DEV_TIMER

```
void        set_period      (device_t *dev, unsigned period);
unsigned    get_value       (device_t *dev);
void        reset           (device_t *dev);
```

DEV_GENERIC

```
int         read            (device_t *dev, dev_request_t *req);
int         write           (device_t *dev, dev_request_t *req);
int         set_param       (device_t *dev, dev_param_t *param);
int         get_param       (device_t *dev, dev_param_t *param);
```

DEV_CPU

-- rien -- (pour le moment)

On suppose que les périphériques sont toujours présents et ils sont initialisés par une fonction **arch_init()** invoquée peu après le reset.
Cette fonction fait appel aux fonctions d'initialisation de chaque périphérique qu'il faudra écrire mais qui ne sont pas exportés pour un usage par les threads.

Que veut-on faire ?

- On veut pouvoir envoyer des requêtes aux périphériques
  - lire ou envoyer des données,
  - démarrer un compteur,
  - programmer un transfert, etc.
- Une requête contient des paramètres et une case de status informant de sa terminaison, elle peut être bloquante ou non pour le thread demandeur.
  - Si les requêtes sont non bloquantes, lorsqu'un thread envoie une requête et que le périphérique est occupé alors la requête est placée dans une file d'attente gérée par le pilote du périphérique. Le thread n'est pas mis en attente, il devra scruter le status la requête.
  - Si les requêtes sont bloquantes, lorsqu'un thread envoie une requête, il est mis à l'état WAIT et ne sera remis à l'état READY qu'une fois la requête terminée.
- On doit accepter les interruptions du périphérique
  - Une interruption informe de la fin d'exécution d'une requête, de la demande de travail à faire, de l'arrivée de données à traiter, etc.
  - L'avancement de la file d'attente des requêtes est géré par le gestionnaire d'interruption du périphérique (irq_handler ou isr pour interrupt service routine)
  - l'exécution du gestionnaire souhaité est réalisé par *un routage des interruptions.*

Requête générique : dev_request_t

Une requête décrit un travail à réaliser pour tout périphérique,
l'interprétation des champs dépend du périphérique.

```
typedef struct dev_request_s {

    // public (renseigné et lu par le kernel)
    void *      src;        // adresse origine des données
    void *      dst;        // adresse destination des données
    unsigned    size;       // nombre de données (en octets ou en blocs)
    int         status;     // code de retour (p.ex. erreur)

    // private (nécessaire pour la gestion de la requête par le périph.)
    list_t      list;       // stockage des requêtes pendantes
    void *      data;       // extension pour la gestion de la requête

} dev_request_t;
```

Paramètre générique : dev_param_t

Les paramètres dépendent des types de périphérique,
Ils servent à la configuration.
L'interprétation des champs dépend du périphérique.
Tous les champs ne pas toujours utiles.

```
typedef struct dev_param_s {
        unsigned type;
        unsigned size;
        unsigned count;
        unsigned speed;
        unsigned xSize;
        unsigned ySize;
} dev_param_t;
```

Driver : driver_t 1/2

```
typedef union driver_u {
    struct dev_cpu_op       cpu;
    struct dev_icu_op       icu;
    struct dev_timer_op     timer;
    struct dev_generic_op   generic;
} driver_t;
```

```
typedef void    (icu_setclr_mask_t)     (device_t * icu, unsigned mask);
typedef unsigned  (icu_get_mask_t)          (device_t * icu);
typedef unsigned  (icu_get_highest_irq_t)(device_t * icu);
struct dev_icu_op {
    icu_setclr_mask_t *         set_mask;
    icu_setclr_mask_t *         clr_mask;
    icu_get_mask_t *            get_mask;
    icu_get_highest_irq_t *     get_highest_irq;
};
```

```
typedef void   (timer_set_period_t) (device_t * timer, unsigned period);
typedef unsigned (timer_get_value_t)  (device_t * timer);
typedef void   (timer_reset_t)          (device_t * timer);
struct dev_timer_op {
    timer_set_period_t *    set_period;
    timer_get_value_t *     get_value;
    timer_reset_t *         reset;
};
```

périphérique : device_t

```
typedef struct device_s device_t;
struct device_s {
    spinlock_t              lock;     // en cas d'usage par + threads
    void *                  base;     // adresse en mémoire physique
    uint_t              irq;        // numéro de ligne INT
    char *                  name;     // utile pour le debug
    dev_type_t          type;     // type du périphérique
    driver_t                op;       // opérations sur le périphérique
    struct irq_action_s     action;   // handler d'interruption
    void *                  data;     // usage dépendant du driver
};
typedef enum {
    DEV_CPU,
    DEV_ICU,
    DEV_TIMER,
    DEV_GENERIC
} dev_type_t;
```

Pour chaque type de périphérique
on a des opérations spécifiques
définies dans les **pilotes**

Il y a une structure device par instance de périphérique (1 par terminal, 1 par DMA, etc).
Les structures sont créées et initialisées au démarrage.

Notez que la structure n'a été opacifiée, mais elle pourrait l'être, il faudrait la déclarer dans un fichier (p.ex. device.c) et écrire des fonctions accesseurs. L'interêt est la possiblité de contrôle de bon usage.

Driver : driver_t 2/2

```
typedef union driver_u {
    struct dev_cpu_op       cpu;
    struct dev_icu_op       icu;
    struct dev_timer_op     timer;
    struct dev_generic_op   generic;
} driver_t;
```

```
typedef int (dev_generic_request_t) (device_t *dev, dev_request_t *req);
typedef int (dev_generic_param_t)   (device_t *dev, dev_param_t *param);

struct dev_generic_op {
    dev_generic_request_t *   read;
    dev_generic_request_t *   write;
    dev_generic_param_t *     set_param;
    dev_generic_param_t *     get_param;
};
```

Déclaration des périphériques

Pour notre plateforme: pour chaque périphérique, on définit un tableau avec autant de cases qu'il y a d'instances.

```
device_t cpu_tbl    [CONFIG_CPU_NR];
device_t icu_tbl    [CONFIG_ICU_NR];
device_t timer_tbl  [CONFIG_TIMER_NR];
device_t tty_tbl    [CONFIG_TTY_NR];
device_t fb_tbl     [CONFIG_FB_NR];
device_t dma_tbl    [CONFIG_DMA_NR];
device_t bd_tbl     [CONFIG_BD_NR];
```

Ces tableaux ne sont pas exportés, on accède aux instances par

```
extern device_t * dev_icu    (unsigned id);
extern device_t * dev_timer  (unsigned id);
extern device_t * dev_tty    (unsigned id);
extern device_t * dev_dma    (unsigned id);
extern device_t * dev_fb(unsigned id);
extern device_t * dev_bd     (unsigned id);
extern device_t * dev_cpu    (unsigned id);
```

TTY

```
void soclib_tty_init(device_t *dev, void * base, char *name)
{
   spinlock_init                (&(dev->lock));
   dev -> base                  = base;
   dev -> irq                   = -1;   // non connecté
   dev -> name                = name;
   dev -> type                  = DEV_GENERIC;
   dev -> op.generic.read       = &tty_read;
   dev -> op.generic.write      = &tty_write;
   dev -> op.generic.set_param= &tty_param;
   dev -> op.generic.get_param= &tty_param;
   dev -> action.dev            = tty;
   dev -> action.irq_handler  = &tty_irq_handler;
   dev -> data                  = NULL;
}
```

Il y a une fonction de ce genre par type de périphériques
et **arch_init()** autant d'appel que d'instances

Initialisation des périphériques

Pour chaque périphérique, il y a une fonction d'initialisation.
Elles seront appelées par la fonction d'initialisation de l'architecture **spécifique à l'architecture**.

```
void arch_init(void) {
   ...
   initialisation de la mémoire;
   ...
   tty_init( tty_tbl+0, TTY_BASE+TTY_SPAN*0, 1, "tty0");
   tty_init( tty_tbl+1, TTY_BASE+TTY_SPAN*1, 2, "tty1");
   ...
}
```

Ces fonctions ne sont pas exportées mais font partie de l'abstraction.
Elles initialisent les structures device_t

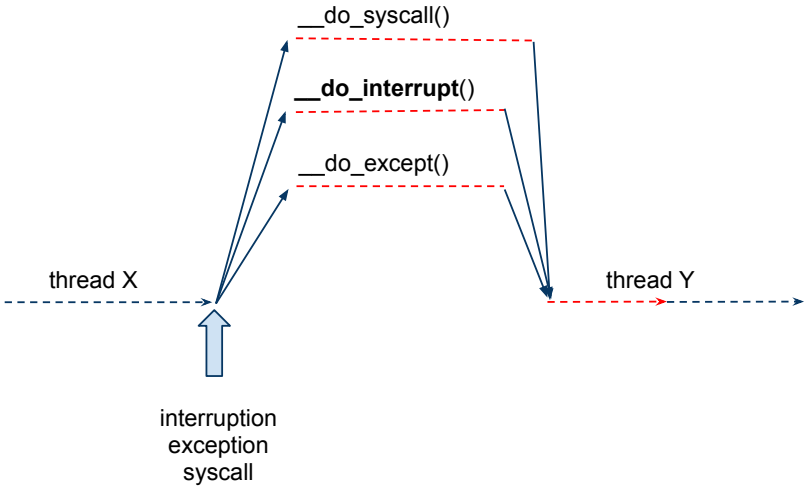
TTY (en mode bloquant / attente active)

```
static int tty_read (device_t * tty, dev_request_t * req) {
   volatile uint32_t * base = tty->base;
   char *dst = req->dst;
   while (base[TTY_STATUS_REG]) return 0;
   *dst = base[TTY_READ_REG];
   return 1;
}
static int tty_write (device_t * tty, dev_request_t * req) {
   volatile uint32_t *base = tty->base;
   char *src = req->src;
   base[TTY_WRITE_REG] = *src;
   return 1;
}
static ssize_t tty_params(struct device_s *dev, dev_params_t * params) {
   kprintf("%s does not have parameters\n", dev->name);
   return 0;
}
static void tty_irq_handler(struct device_s *dev) {
   volatile uint32_t *base = dev->base;
   char c = base[READ_REG];
   base[WRITE_REG] = c;
}
```

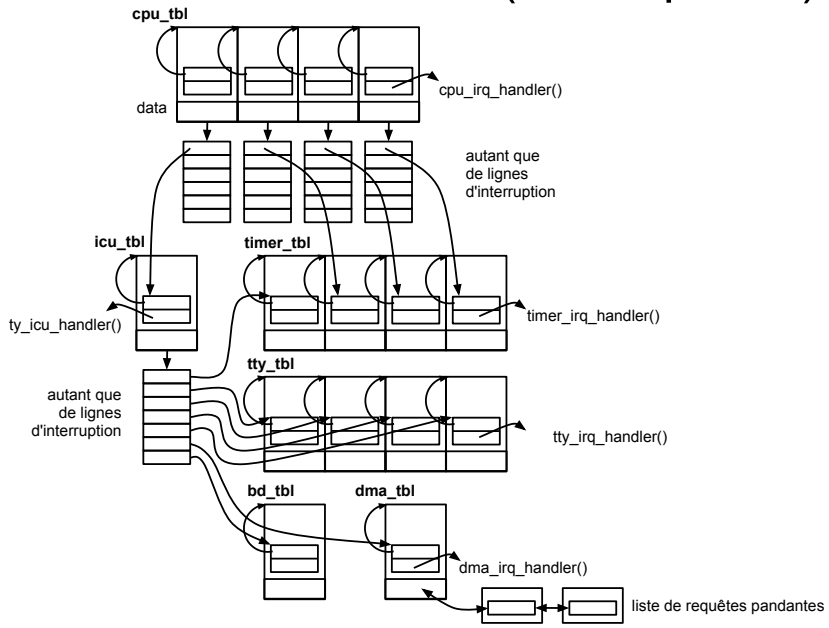
Exemple d'usage du tty

```
void fputs(unsigned tty_id, char *buffer)
{
   device_t *tty = dev_tty(tty_id);
   struct dev_request_s req;
   unsigned size, written_char=0;

   for(size=0; buffer[size]; size++);

   spin_lock(&(tty->lock));
   while (size != written_char) {
      req.src = buffer + written_char;
      req.size = size - written_char;
      written_char += tty->op.generic.write(tty, &req);
   }
   spin_unlock(&(tty->lock));
}
```

Les points d'entrée du noyau

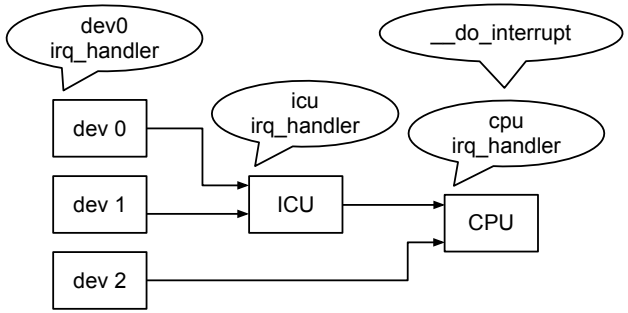


Liaison entre les devices (interruptions)



Les interruptions

Un périphérique qui lève une interruption veut que ce soit le gestionnaire initialisé dans le driver qui s'exécute.



Routage des interruptions

Un périphérique crée un événement et lève une interruption.
l'interruption est routée physiquement jusqu'au CPU en passant par l'ICU.
On suppose qu'elle n'est pas masquée.

Le processeur est sollicité, il exécute :

- kentry
  - sauve les registres temporaires (pas les persistants car sauvés par gcc)
  - appelle __do_interrupt qui appelle le handler du CPU
  - restaure les registres temporaires

```
void __do_interrupt(unsigned cpu_id) {
    device_t *cpu = dev_cpu(cpu_id);
    cpu->action.irq_handler(cpu->action.dev);
}
```

action pour les interruptions

Associe un périphérique et un handler d'interruption (ISR)

```
// couple pointeur sur structure device_s / handler
struct irq_action_s {
    device_t *        dev;          // pointeur sur le périphérique
    irq_handler_t *   irq_handler;  // pointeur sur la fonction gestionnaire
}

// pointeur sur fonction
typedef void (irq_handler_t) (struct irq_action_s *action);
```

CPU (détails)

```
void cpu_init(struct device_s *dev, void * base, char *name) {
    spin_init(&dev->lock);
    dev->base           = base;
    dev->irq            = -1;
    dev->name           = name;
    dev->type           = DEV_CPU;
    dev->action.dev     = dev;
    dev->action.irq_handler = &cpu_irq_handler;
    dev->data           = kmalloc(sizeof(struct irq_action_s *) * CONFIG_CPU_MAX_IRQ_NR);
}

void cpu_bind(struct device_s *cpu, struct device_s *dev, uint_t irq) {
    struct irq_action_s **action_tbl = cpu->data;
    dev->irq = irq;
    action_tbl[irq] = &(dev->action);
}

static void cpu_irq_handler (struct device_s *dev) {
    struct irq_action_s **action = dev->data;
    int irq = cpu_get_highest_irq();
    action[irq]->irq_handler(action[irq]->dev);
}
```

ICU

Fonctions appelées par arch_init()

icu_init(device_t *icu, void * base, char *name);

- initialise : lock base, irq, name, type, op (driver), irq_action
- initialise la structure de travail : data

icu_bind(device_t *icu, device_t *dev, unsigned irq);

- fait un lien entre le périphérique de l'icu et un autre périphérique

Fonctions appelées par le driver

icu_set_mask(...)
icu_get_mask(...)
icu_get_highest_irq(...)
icu_irq_handler(...)

```
void
icu_init(device_t *dev, void * base, char *name)
{
    spin_init(&dev->lock);
    dev->base                   = base;
    dev->irq                    = -1;
    dev->name                   = name;
    dev->type                   = DEV_ICU;
    dev->op.icu.set_mask        = &icu_set_mask;
    dev->op.icu.clr_mask        = &icu_clr_mask;
    dev->op.icu.get_mask        = &icu_get_mask;
    dev->op.icu.get_highest_irq = &icu_get_highest_irq;
    dev->action.dev             = dev;
    dev->action.irq_handler     = &icu_irq_handler;
    dev->data = kmalloc(sizeof(struct irq_action_s *)
                        * CONFIG_ICU_MAX_IRQ_NR);
}
```

ICU

- pour l'icu le champ data sera simplement un tableau de pointeur irq_action_s avec autant de cases qu'il y a d'interruptions reçues par l'ICU.

- L'opération bind a pour effet d'initialiser la case de ce tableau dont le numero est passé en paramètre.
  Le champ irq du périphérique est mis à jour.

```
icu_bind(device_t *icu, device_t *dev, unsigned_t irq) {
      struct irq_action_s **action = icu->data;
      dev->irq = irq;
      action[dev->irq] = &(dev->action);
}
```

- Le handler d'interruption est semblable à celui du CPU

```
static void icu_irq_handler (struct device_s *dev) {
   struct irq_action_s **action = dev->data;
   int irq = icu_get_highest_irq(dev);
   action[irq]->irq_handler(action[irq]->dev);
}
```

handler du timer

Au plus simple on va supposer que le timer est programmé pour provoquer une interruption périodique, et qu'à chaque période, le système demande une commutation de thread.

donc de manière périodique: on appelera
    timer_irq_handler(device_t * dev)

où l'on fait deux choses:
- reset pour acquitter l'interruption
- sched_clock

Initialisation de l'architecture

En résumé:

__do_init va appeler une fonction **arch_init** définie dans le répertoire arch/soclib

cette fonction va demander
- l'initialisation des périphériques et du cpu
- faire des liens pour les handlers avec les fonctions bind

Gestion des requêtes : DMA

```
#include <config.h>
#include <list.h>
#include <thread.h>
#include <libk.h>

// ---------------------------------------------------------------------------------------------------
// DEVICE REGISTER MAP
// ---------------------------------------------------------------------------------------------------

#define DMA_SRC_REG          0
#define DMA_DST_REG          1
#define DMA_LEN_REG          2
#define DMA_RESET_REG        3
#define DMA_IRQ_DISABLED_REG 4

struct pending_req_s {
    list_t req;         // pending requests for the device
    list_t thread;      // pending threads for the device
};

// ---------------------------------------------------------------------------------------------------
// DRIVER OPERATION
// ---------------------------------------------------------------------------------------------------

static void dma_start_req(struct device_s *dev, struct dev_request_s *req)
{
    MESSG(DEBUG,"%s execute req %p", dev->name, req);
    volatile uint32_t *base = dev->base;
    base[DMA_IRQ_DISABLED_REG] = 1;
    base[DMA_SRC_REG] = (uint32_t)req->src;
    base[DMA_DST_REG] = (uint32_t)req->dst;
    base[DMA_LEN_REG] = (uint32_t)req->size;
}
```

Gestion des requêtes : DMA

```
static ssize_t dma_write(struct device_s *dev, struct dev_request_s *req)
{
    uint_t status;
    struct pending_req_s *pending = dev->data;

    MESSG(DEBUG,"add req %d for thread %p", req, sched_thread_run());

    spin_lock(&dev->lock);                      // then no one else uses this device
    cpu_disable_all_irq(&status);               // then no irq possible

    if (list_isempty(&pending->req))            // if no pending request, then command the device
        dma_start_req(dev, req);

    list_add_last(&pending->req, &req->list);   // Always register the req to the pending request
    thread_list_add_last(&pending->thread, sched_thread_run()); // and register the thread too

    // the thread is always running but it is already waiting for the dev in the pending req list
    spin_unlock(&dev->lock);    // device may accept a new request but can't finish the current one
    sched_sleep();              // blocking mode by default, sleep and schedule

    cpu_restore_irq(status);    // restore the mode
    return 0;
}

static ssize_t dma_read(struct device_s *dev, struct dev_request_s * req)
{
    return dma_write(dev,req);
}

static ssize_t dma_params(struct device_s *dev, dev_params_t * params)
{
    ERROR("%s does not have parameters", dev->name);
    return 0;
}
```

Gestion des requêtes : DMA

```
// ---------------------------------------------------------------------------------------------------
// DEVICE INITIALIZATION
// ---------------------------------------------------------------------------------------------------

void dma_init(struct device_s *dev, void *base, char *name)
{
    spin_init(&dev->lock);
    dev->base = base;
    dev->name = name;
    dev->irq = -1; // not bound yet
    dev->type = DEV_GENERIC;
    dev->op.generic.read = &dma_read;
    dev->op.generic.write = &dma_write;
    dev->op.generic.set_params = &dma_params;
    dev->op.generic.set_params = &dma_params;
    dev->action.dev = dev;
    dev->action.irq_handler = &dma_irq_handler;
    struct pending_req_s *pending = kmalloc(sizeof(struct pending_req_s));
    list_root_init(&pending->req);
    list_root_init(&pending->thread);
    dev->data = pending;
}
```

Gestion des requêtes : DMA

```
static void dma_irq_handler(struct device_s *dev)
{
    volatile uint32_t *base = dev->base;
    struct pending_req_s *pending = dev->data;

    spin_lock(&dev->lock);              // then no one else uses this device

    // unlink the pending request since the irq means it is finished
    struct dev_request_s *req = list_item(list_unlink(list_first(&pending->req)),
                                          struct dev_request_s, list);
    req->err = base[DMA_LEN_REG];       // 0 means, there are no error
    base[DMA_RESET_REG] = 0;            // irq acknowledgment

    thread_t *thread = thread_list_item(list_unlink(list_first(&pending->thread)));
    sched_wakeup(thread);               // wakeup the pending thread

    if (! list_isempty(&pending->req))  // if there is another waiting request, then start it
        dma_start_req(dev, list_item(list_first(&pending->req), struct dev_request_s, list));

    spin_unlock(&dev->lock);            // device may accept a new request
}
```