Gestion des périphériques

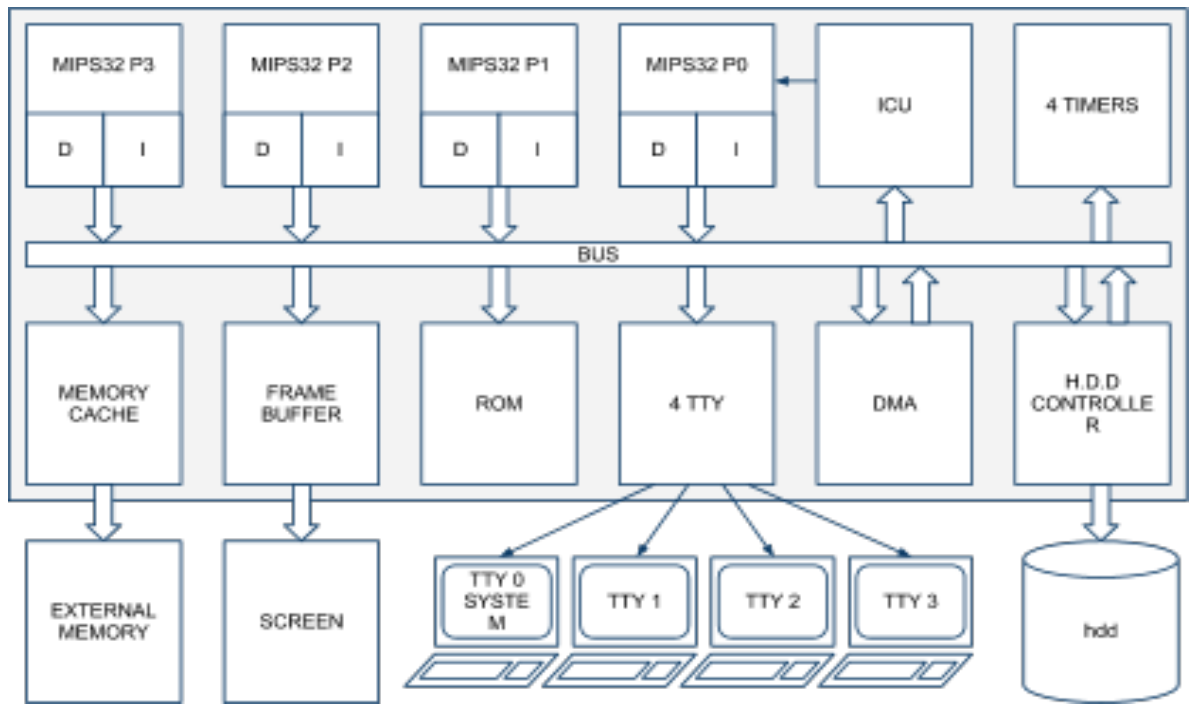
gestion de l'architecture

MI074 - 6

Les périphériques / devices

- Un périphérique 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.



Le processeur (cpu)

- Un processeur exécute des threads.
- Un thread est un programme en cours d'exécution défini par:
  - un état : prêt, bloqué, ...
  - un code de programme.
  - un contexte d'exécution (état du processeur).
  - un pile locale.
  - un état global partagé par les autres threads.
- Le processeur peut être vu comme un type particulier de device, à qui on demande d'exécuter des threads.
- C'est un device défini par le kernel

Abstraction de l'architecture

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

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

- On va définir une collection de fonctions (une API) pour chaque type de device dont:

  - le prototype est imposé par le noyau mais dont
  - l'implémentation est spécifique à chaque type de device

L'ensemble de ces fonctions constitue le driver du device

Que veut-on faire ?

- On veut pouvoir envoyer des requêtes aux devices (requests)
  - lire des données
  - envoyer des données
  - programmer un transfert
  - démarrer un compteur
  - endormir
  - réveiller
  - etc

  - Lorsqu'on envoie une requête et que le device est occupé, la requête est placée dans une file d'attente gérée par le pilote (driver) du device.

- On doit accepter les interruptions du device

  - 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 device (irq_handler ou isr pour interrupt service routine)
  - l'exécution de la bonne fonction gestionnaire est réalisé par un routage des interruptions.

Device : device_s

```
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 device
    driver_t                op;     // opérations sur le device
    struct irq_action_s     action; // handler d'interruption
    void *                  data;  // usage dépendant du driver
};

typedef enum {
    DEV_ICU,
    DEV_TIMER,
    DEV_GENERIC
} dev_type_t;
```

Pour chaque type de device
des opérations spécifiques
définies dans les **drivers**

Device CPU

Le CPU est parfois vu comme un device,
mais il appartient et est géré par le kernel

```
struct cpu_s {
    spinlock_t              lock;    // protection accès parallèle
    char *                  name; // utile pour le debug
    struct thread_s *       run;     // thread en cours
    struct thread_s *       idle;    // thread tâche de fond
    list_t                  ready;  // liste des threads en attente
    list_t                  dead;   // liste des threads morts
    struct sched_op         op;      // operations du scheduler
    void *                  data;    // usage dépendant du driver
};
```

Opérations sur les devices

DEV_ICU
void set_mask(struct device_s *dev, uint_t mask);
uint_t get_mask(struct device_s *dev);
uint_t get_highest_irq(struct device_s *dev);

DEV_TIMER
void set_period(struct device_s *dev, uint_t period);
uint_t get_value(struct device_s *dev);

DEV_GENERIC
ssize_t read(struct device_s *dev, **dev_request_t** *req);
ssize_t write(struct device_s *dev, **dev_request_t** *req);
ssize_t set_param(struct device_s *dev, **dev_param_t** *req);
ssize_t get_param(struct device_s *dev, **dev_param_t** *req);

Requête générique

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

```
typedef struct dev_request_s {
// public (renseigné et lu par le kernel)
void *    src;        // adresse origine des données
void *    dst;        // adresse destination des données
size_t    size;       // nombre de données (en octets ou en blocs)
int       error;      // code d'erreur
// private (nécessaire pour la gestion de la requête par le device)
list_t    list;       // stockage des requêtes pendantes
void *    data;       // extension pour la gestion de la requête
} dev_request_t;
```

Paramètre générique

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

```
typedef struct dev_param_s {
        uint32_t type;
        uint32_t size;
        uint32_t count;
        uint32_t speed;
        uint32_t xSize;
        uint32_t ySize;
    } dev_param_t;
```

Driver : driver_t 1/2

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

```
typedef void (icu_set_mask_t) (struct device_s * icu, uint_t mask);
typedef uint_t (icu_get_mask_t) (struct device_s * icu);
typedef uint_t (icu_get_highest_irq_t) (struct device_s * icu);

struct dev_icu_op {
    icu_set_mask_t *set_mask;
    icu_get_mask_t *get_mask;
    icu_get_highest_irq_t *get_highest_irq;
};

typedef void (timer_set_period_t) (struct device_s * timer, uint_t period);
typedef uint_t (timer_get_value_t) (struct device_s * timer);

struct dev_timer_op {
    timer_set_period_t *set_period;
    timer_get_value_t *get_value;
};
```

Driver : driver_t 2/2

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

```
typedef ssize_t (dev_generic_request_t)
                    (struct device_s *dev, dev_request_t *req);
typedef ssize_t (dev_generic_param_t)
                    (struct device_s *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 devices

Pour chaque device, on définit un tableau
avec autant de cases qu'il y a d'instances.

```
struct device_s icu_tbl[CONFIG_ICU_NR];
struct device_s timer_tbl[CONFIG_TIMER_NR];
struct device_s tty_tbl[CONFIG_TTY_NR];
struct device_s fb_tbl[CONFIG_FB_NR];
struct device_s dma_tbl[CONFIG_DMA_NR];
struct device_s bd_tbl[CONFIG_BD_NR];
```

Initialisation des devices

Pour chaque device, on 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;
    ...
    soclib_tty_init( tty_tbl+0, TTY_BASE+TTY_SPAN*0, 1, "tty0");
    soclib_tty_init( tty_tbl+1, TTY_BASE+TTY_SPAN*1, 2, "tty0");
    ...
}
```

Le tty

```
void soclib_tty_init(struct device_s *tty, void * base, uint_t irq)
{
    spinlock_init(&tty->lock);
    tty->base = base;
    tty->irq = irq;
    tty->type = DEV_GENERIC;
    tty->op.generic.read = &soclib_tty_read;
    tty->op.generic.write = &soclib_tty_write;
    tty->op.generic.set_param = NULL;
    tty->op.generic.get_param = NULL;
    tty->action.dev = tty;
    tty->action.irq_handler = &soclib_tty_irq_handler;
    tty->data = NULL;
}
```

Le tty (mode bloquant)

```
static ssize_t tty_read (struct device_s * 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 ssize_t tty_write (struct device_s * tty, dev_request_t * req)
{
    volatile uint32_t *base = tty->base;
    char *src = req->src;
    base[TTY_WRITE_REG] = *src;
    return 1;
}
```

Exemple d'usage du tty

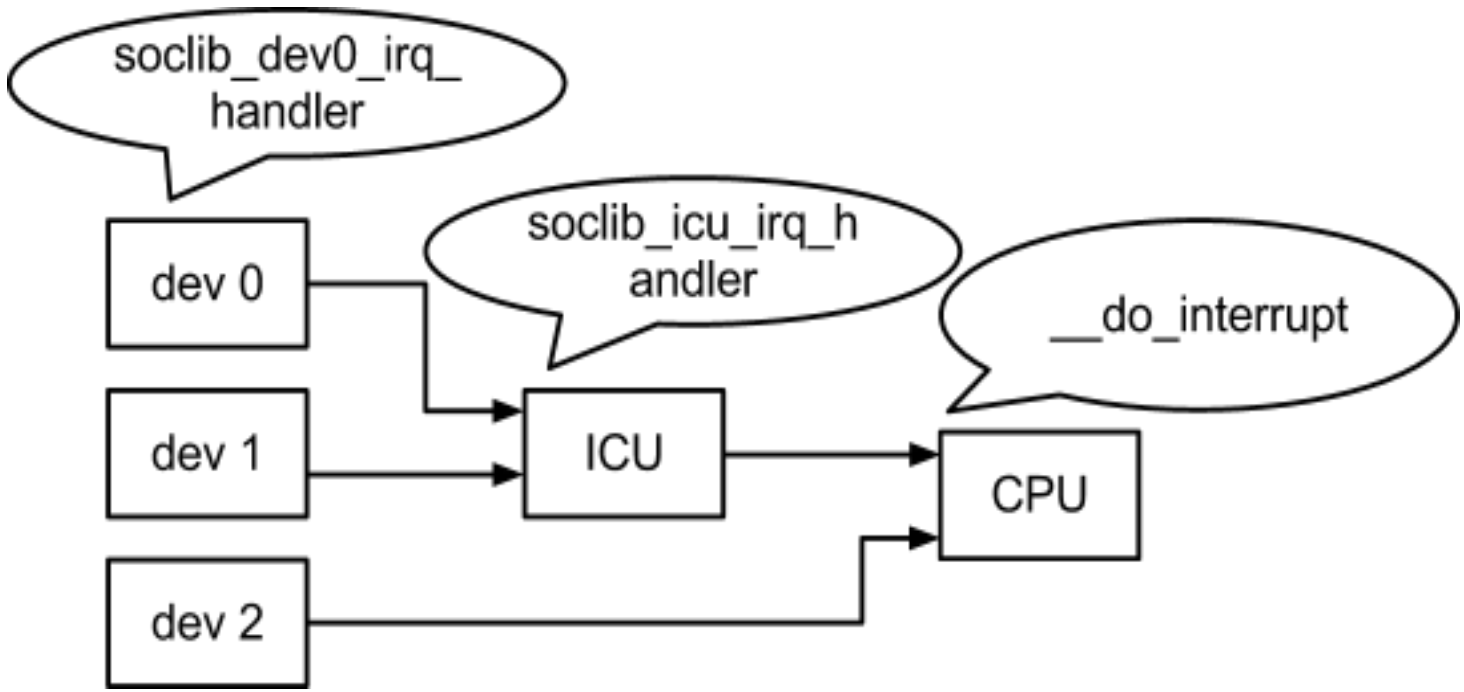
```
void fputs(uint_t tty_id, char *buffer)
{
    struct device_s *tty = &tty_tbl[tty_id];
    struct dev_request_s req;
    uint_t size, written_char=0;

    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 interruptions

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

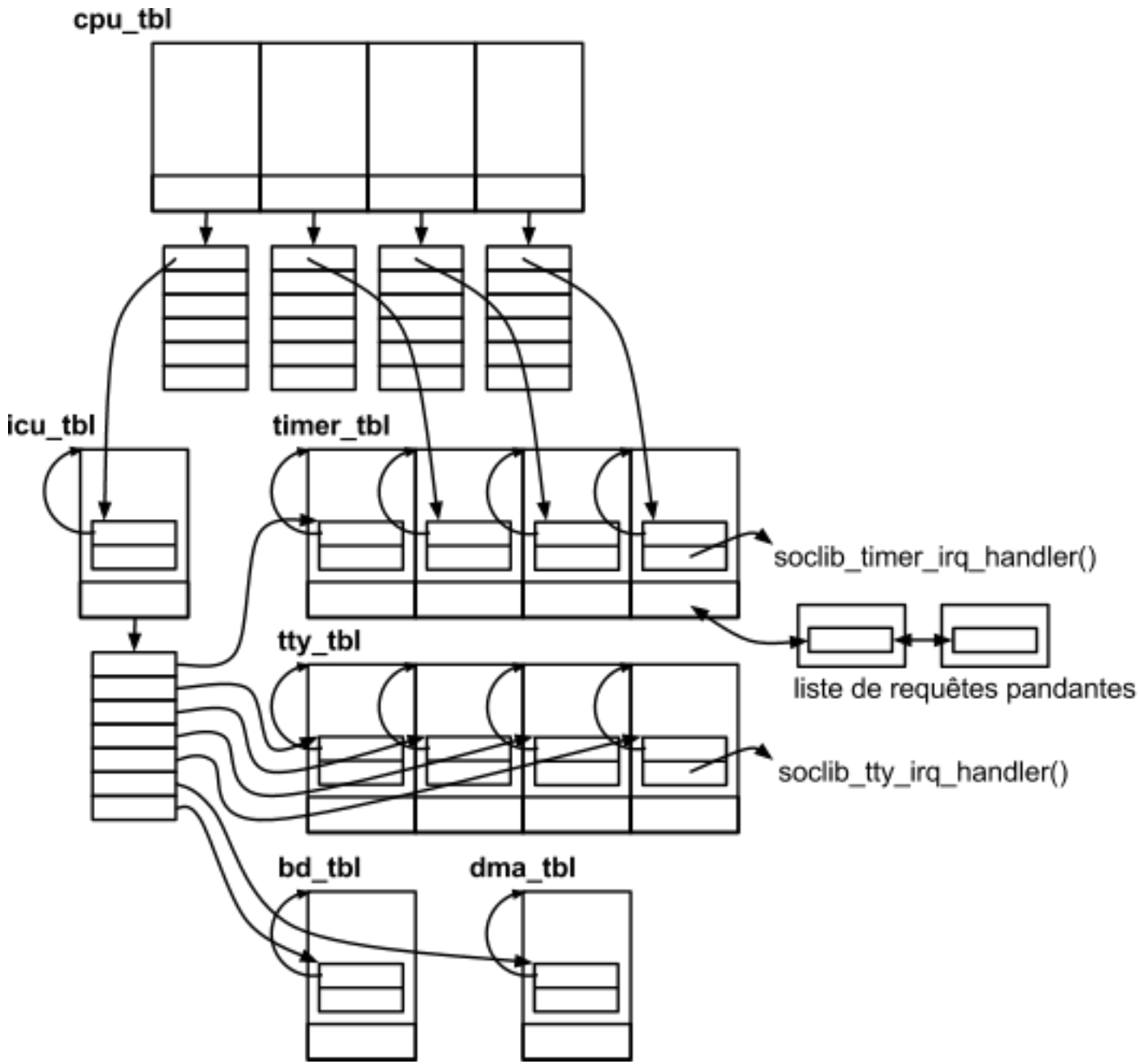


action pour les interruptions

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

```
    // couple pointeur sur structure device_s / handler
struct irq_action_s {
    struct device_s * dev;
    irq_handler_t *     irq_handler;
}
    // pointeur sur fonction
typedef void (irq_handler_t) (struct irq_action_s *action);
```

Vue de l'architecture



ICU

Fonctions appelées par arch_init()
icu_init(struct device_s *icu, void * base, uint_t irq);

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

icu_bind(struct device_s *icu, struct device_s *dev);

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

Fonctions appelées par le driver
icu_set_mask(...)
icu_get_mask(...)
icu_get_highest_irq(...)
icu_irq_handler(...)

Cas d'étude : ICU

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

- L'opération bind a pour effet d'initialiser la case de ce tableau dont le numero se trouve dans le device lui-même

```
struct irq_action_s **action = icu->data;
action[dev->irq] = &(dev->action);
```

Routage des interruptions

Un device 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 interrompu, il exécute :

- kentry
- __do_interrupt qui agit comment le handler irq du CPU
  - l'argument de do_interrupt est égal à l'état des lignes IRQ
  - on va chercher dans le tableau data l'action à exécuter et on l'exécute :

```
action = cpu->data[highest(irqstate)];
action->irq_handler(action->dev);
```

Initialisation de l'architecture

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

cette fonction va demander l'initialisation des devices et du cpu puis faire des liens pour les handler avec les fonctions bind

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(struct device_s * timer)

où l'on fait deux choses:
reset_irq

et sched.yield (lorsqu'on aura les threads)
pour le moment un affichage.