```
void __init stm32_button_init(void)
{
  int rv;
  printk("INFO: enabling interrupt on PA0 (%d/%d)\n",
                STM32429_DISCO_BUTTON_EXTILINE,
                STM32F2_EXTI_NUM_LINES);

  stm32_exti_enable_int(STM32429_DISCO_BUTTON_EXTILINE,1);

  rv = request_irq(STM32429_DISCO_BUTTON_INT,
                stm32_bouton_isr,
                IRQF_TRIGGER_RISING,
                "STM32 TEXI0/PA0 Button",
                (void *)stm32_bouton_isr);
  if(rv) {
        printk("INFO: %s: request_irq(%d) failed (%d)\n",
                        __func__, STM32429_DISCO_BUTTON_INT, rv);
  } else {
        printk("INFO: BOUTON interrupt handler installed \n");
  }
}
```

Nous avons là une installation classique d'ISR dans Linux via **request_irq()**. Notez qu'il est impératif de tester la valeur retournée indiquant la bonne marche de l'installation (on ne peut, par exemple définir deux ISR sur une même interruption). Autre élément important, l'utilisation de la fonction **stm32_exti_enable_int()** décrite précédemment et permettant de configurer matériellement la génération d'interruption.

Le fichier **uclinux/arch/arm/mach-stm32/include/mach/button.h** contiendra simplement les prototypes de nos deux fonctions. Nous n'oublions pas de modifier **stm32_platform.c** afin d'y inclure un appel à **stm32_button_init()** et de modifier **uclinux/arch/arm/mach-stm32/Makefile** pour inclure notre **button.o** à **obj-$(CONFIG_MACH_STM32)**.

Après compilation et enregistrement du noyau dans la flash de la carte, nous avons le plaisir de constater que notre routine est bien installée :

```
INFO: enabling interrupt on PA0 (0/23)
INFO: BOUTON interrupt handler installed
```

Nous en aurons confirmation en jetant un simple coup d'oeil dans :

```
~ # cat /proc/interrupts
           CPU0
  6:          1   NVIC  STM32 TEXI0/PA0 Button
28:     361030   NVIC  STM32 Kernel Tine Tick
37:        500   NVIC  STM32 USART Port
60:          0   NVIC  STM32 USART Port
Err:         0
```

Et bien entendu, une pression sur le bouton provoquera le comportement attendu et l'affichage du message de la part du noyau :

```
INFO: irq on blue button !
```

Je ne sais pas ce que vous en pensez, mais je suis relativement mal à l'aise vis-à-vis de ce code. Dans l'état, nous ne pouvons utiliser cette gestion que de manière extrêmement limitée. Il nous est, par exemple, impossible de choisir le port utilisé pour une ligne donnée. Ici, nous sommes coincé sur le port A aussi bien pour la broche 0 que n'importe quel autre GPIO encore disponible sur le devkit. EmCraft n'a implémenté aucun mécanisme permettant de configurer les multiplexeurs de l'EXTI et nous allons tâcher de corriger ce problème.

2 Faisons un peu plus propre

Nous allons donc réviser le contenu du fichier **exti.c** et l'adapter à nos besoins tout en faisant un peu le ménage dans les désignations utilisées. Commençons par revoir la fonction effaçant le drapeau d'interruption :

```
int stm32_exti_clear_pending(unsigned int line)
{
        if (line >=  STM32F2_EXTI_NUM_LINES)
                return -EINVAL;

        STM32_EXTI->pr = (1 << line);

        return 0;
}
EXPORT_SYMBOL(stm32_exti_clear_pending);
```

Rien de bien méchant pour l'instant. Nous testons **line** comme précédemment mais avec une logique inversée (je n'aime pas beaucoup les « un coup <, un coup >= ») et une fonction retournant un **int** permettant de tester son bon fonctionnement. La macro et la structure associée a également été révisée de manière à rester logique et nous défaire de la mention d'un autre microcontrôleur pouvant semer une certaine confusion :

```
/*
* EXTI register map base
*/
struct stm32_exti_regs {
  u32 imr;          /* Interrupt mask register */
  u32 emr;          /* Event mask register */
  u32 rtsr;         /* Rising trigger selection register */
  u32 ftsr;         /* Falling trigger selection register */
  u32 swier;        /* Software interrupt event register */
  u32 pr;           /* Pending register */
};
#define STM32_EXTI_BASE 0x40013c00
#define STM32_EXTI      ((volatile struct stm32_exti_regs *) \
                STM32_EXTI_BASE)

/*
* SYSCFG register map base
*/
struct stm32_syscfg_reg {
        u32 memrm;
       u32 pmc;
        u32 exticr1;
        u32 exticr2;
        u32 exticr3;
        u32 exticr4;
        u32 cmpcr;
};
#define SYSCFG_BASE     0x40013800
#define SYSCFG          ((volatile struct stm32_syscfg_reg *) \
                SYSCFG_BASE)
```

Nous en profitons pour ajouter le même type de mécanisme pour accéder aux registres de configuration **SYSCFG***. Nous