FOX DOC

28 ottobre 2007
Linux Day

Sommario

Scopo del documento è cercare di avvicinare chi legge al mondo Linux Embedded andando a focalizzare l'attenzione sulla FOX board in quanto è un prodotto completo sia in termini di hardware che software ed inoltre è uno strumento di relativamente semplice utilizzo, versatile e completamente libero.

Questo documento riassume l'esperienza fatta su questa scheda fatta dal sottoscritto (e chiaramente da tutti coloro che vorranno aricchire il documento). Gli argomenti trattati sono generali e vanno dall'hardware al software user/kernel space.

Attualmente l'esperienza riportata riguarda solo l'utilizzo delle porte GPIO del processore AXIS ETRAX 100LX a cui sono collegati i LED (compresi quelli del connettore ethernet) e il pulrante.

1 Introduzione all'ETRAX 100LX

Prima di iniziare è bene avere alcune informazioni relative al processore le quali si possono trovare in modo più dettagliato sul datasheet dello stesso. Per ora basti sapere che l'ETRAX 100LX è un processore RISC a 32bit da 100MIPS[1] con 8kbyte interni per la cache. Tutte le periferiche e controlli sono *memory mapped*, ossia esiste almeno un indirizzo fisico di memoria (registro) per ogni periferica nel quale ogni bit assume un particolare significato relativamente al funzionamento della periferica stessa. Ciò è molto comodo perchè permette di gestire le periferiche in modo semplice e diretto senza ricorrere a particolari istruzioni. Esempi di processori non memory mapped sono quelli appartenenti alla famiglia x86.

Memoria L'indirizzamento virtuale è pari a 4Gb e l'accesso alla memoria esterna è gestito da controller integrati nel chip e consente di interfacciarsi a SDRAM, EDO DRAM, SRAM, EPROM, EEPROM, e Flash PROM senza l'uso di dispositivi esterni. É prevista una unità MMU.

I/O 2 Porte seriali sincrone e 4 asincrone, 2 porte parallele IEEE1284 compatibili, SCSI-2 e SCSI-3, IDE/ATA-2, 2 USB (sia host che device!), Ethernet 10/100 fullduplex.

2 GPIO

L'acronimo GPIO indica *General Purpose Input/Output*, ossia dispositivi di ingresso uscita di uso generale. Questi dispositivi detti porte sono fondamentalmente due indicate con A e B[2] composte da 8 bit per ciascuna. A queste porte corrispondono otto pin sul processore il cui stato e tipologia di funzionamento sono indicati da altrettanti bit in appositi registri. Queste porte sono di tipo digitale e vengono dette tri-state in quanto, agendo su un particolare registro a 8 bit (R_PORT_PA_DIR per la porta A), è possibile configurare il singolo pin come input (0) o come output (1). Se è un output agendo su un altro registro (R_PORT_PA_SET per la porta A) sarà possibile portare allo stato alto (1≡5V) o allo stato basso (0≡0V) il singolo pin.

In figura 1 è riportato lo schematico di connessione di alcune porte sulla scheda FOX. Come si nota vi sono tre LED di cui uno sempre acceso (verde) che indica la presenza di tensione, mentre gli altri due (giallo e rosso) sono connessi direttamente al pin 3 (PA2) e 4 (PA3) del micro-processore. Il pulsante (switch SW1) è invece connesso alla porta 2 (PA1).

[1] Mega Instruction Per Second
[2] Esiste anche una terza porta G di cui per il momento non si parlerà

Come si può notare i led sono connessi in logica inversa, ossia l'anodo del diodo led è connesso (tramite una resistenza) a +5V, mentre il catodo è connesso direttamente sul pin del processore. Ciò implica che per accendere un led occorrerà porre a 0 (ossia allo stato logico basso) il pin corrispondente della porta a cui è connesso il led stesso. Analogamente lo switch porterà allo stato logico basso il pin corrispondente a PA1 quando verrà premuto, mentre in stato di riposo (ossia a circuito aperto) il pin sarà posto nello stato logico alto. Questa scelta è tipicamente dovuta al fatto che in fase di conduzione del diodo, la corrente entra nel pin corrispondente; questo pin, però, è a stato logico basso ossia circa 0V dal punto di vista elettrico. Questo implica che la potenza dissipata dal pin del micro-processore è

$$P = V \cdot I \simeq 0$$

e per questo è possibile avere una erogazione di corrente pari anche alla nominale per ogni pin[3] e comunque di ridurre il surriscaldamento del procesore. Si osservi che se i diodi fossero controllati in modo diretto e se supponessimo di erogare 12mA, la potenza dissipata sarebbe per ogni singolo pin

$$P = V \cdot I = 5 \cdot 12 \cdot 10^{-3} = 60mW$$

Questo circuito ha però un difetto: il pulsante infatti porta direttamente a terra il pin della processore. Ora se il pin fosse configurato come uscita e fosse stato settato allo stato logico alto (+5V), alla pressione del tasto si avrebbe un corto-circuito che normalmente causa o la perdita del pin o peggio il danneggiamento totale irreversibile del processore. Normalmente è buona norma porre una resistenza tra lo switch e il pin PA1.

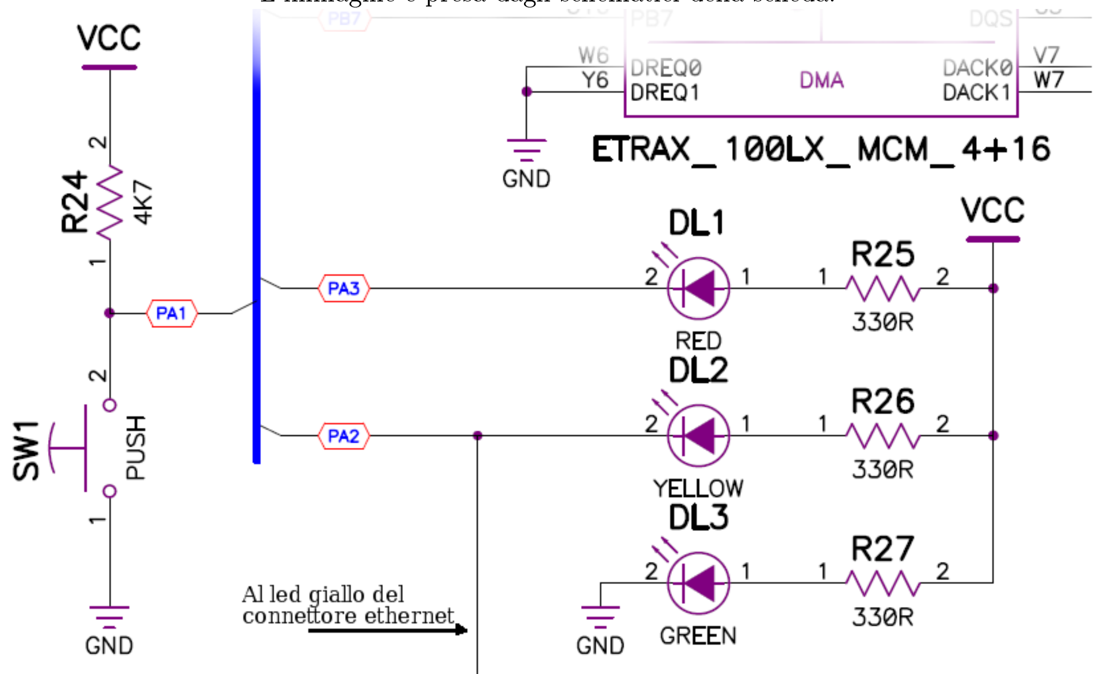
2.1 Configurazione del kernel

La configurazione di default delle porte `A` e `B` avviene a livello kernel. Prendendo in considerazione il kernel 2.6, spostandosi nella sottodirectory `os/linux-2.6/` si può eseguire il tipico comando `make menuconfig`; se invece si preferisce controllare tutto dalla directory principale (tipicamente `devboard-R2_01`) si può eseguire il comando `make kernelconfig`. Ad ogni modo compare la tipica videata testuale di configurazione del kernel.

La configurazione dell'hardware "specifico" per questo processore come le porte GPIO è nella sezione **Hardware setup** riportata in figura 2. É possibile settare quattro valori i quali permettono di indicare

[3] La corrente nominale per pin è di 12mA, ma tipicamente è sempre bene che sia la minore possibile per evitare il surriscaldamento del processore. È buona norma sapere anche qual è la massima potenza dissipabile in totale dal processore.

Figura 1: Connessione di Pulsante e Leds sulla fox board. L'immagine è presa dagli schematici della scheda.



che i LED sono connessi alla porta `A`, oppure alla `B`; quest'ultima però ha delle periferiche condivise e le funzionalità ulteriori (come SCSI e USB) sono prioritarie rispetto alla funzione di I/O generica. `CSP0` (*Cable Selected Port*) va ad elaborare le funzionalità della porta `B` indicando appunto se la funzionalità del singolo pin (dal 2 al 7) è "speciale" o semplicemente un I/O digitale. Scegliere una di queste prime tre opzioni causa la comparsa di altre voci che permettono di settare esattamente a quale pin sono connessi i led.

L'ultima funzione (*None*, indicata anche in figura 2) setta semplicemente in funzione delle opzioni `R_PORT_PA_DIR`, `R_PORT_PA_DATA`, `R_PORT_PB_CONFIG`, `R_PORT_PB_DIR`, `R_PORT_PB_DATA` indicate subito sotto. Queste opzioni definiscono l'esatto valore da porre nel registro specifico e sono presenti anche nelle tre opzioni precedenti, ma non hanno effetto se una di queste opzioni è stata scelta. Prendiamo ora in considerazione la porta `A`[4]: di default è indicato che `R_PORT_PA_DIR=0x1D` e `R_PORT_PA_DATA=0xF0`. Esaminando il valore in bit si ha che `0x1D=0b00011101` e `0xF0=0b11110000` quindi si ha che la porta PA1 è configurata come input[5], mentre PA0, PA2 (LED giallo), PA3 (LED rosso) e PA4 sono output. Questi ultimi sono tutti accesi in quanto i primi quattro bit di `R_PORT_PA_DATA` sono 0.

2.2 Device driver

Il driver in esame è `$AXIS_KERNEL_DIR/arch/cris/arch-v10/drivers/gpio.c` ed è relativo alla gestione di tutte le porte GPIO. Il major number del modulo è assegnato a 120 definito dalla macro

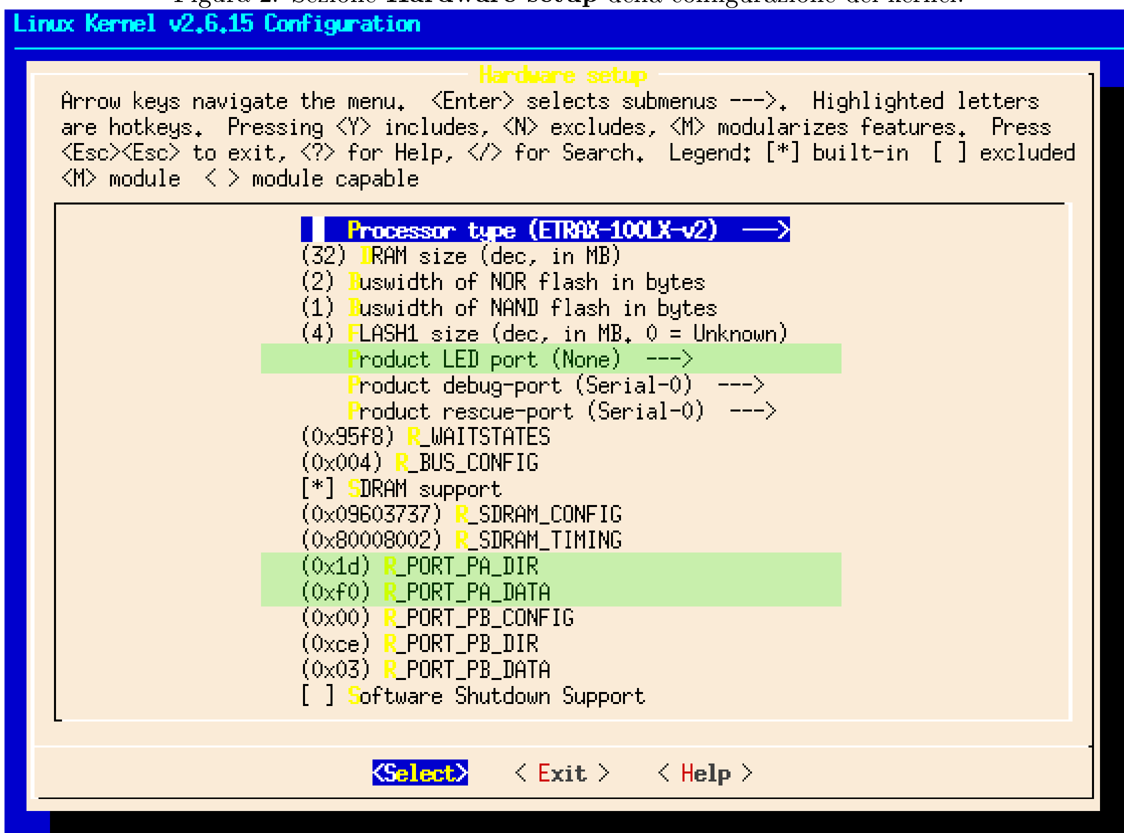
```
#define GPIO_MAJOR 120
```

con minor number 0 e 1 per le porte A e B. Il modulo implementa le seguenti funzioni di I/O per il file di dispositivo:

```
static int gpio_ioctl(struct inode *inode, struct file *file,
                      unsigned int cmd, unsigned long arg);
```

[4]Nella FOX8+32 in esame i LED sono connessi alla porta `A`.
[5]Come indicato, di default viene settato il pin 2 come input per evitare problemi come detto al paragrafo 2.

Figura 2: Sezione **Hardware setup** della configurazione del kernel.



```
static ssize_t gpio_write(struct file * file, const char * buf,
                          size_t count, loff_t *off);
static int gpio_open(struct inode *inode, struct file *filp);
static int gpio_release(struct inode *inode, struct file *filp);
static unsigned int gpio_poll(struct file *filp,
                              struct poll_table_struct *wait);
```

le quali sono passate alla struttura delle "operazioni sui files":

```
struct file_operations gpio_fops = {
        .owner       = THIS_MODULE,
        .poll        = gpio_poll,
        .ioctl       = gpio_ioctl,
        .write       = gpio_write,
        .open        = gpio_open,
        .release     = gpio_release, };
```

Lo sviluppo delle chiamate principali è gestito in modo abbastanza standard. Per cominciare vi è una struttura specifica per la gestione del device chiamata `struct gpio_private` la quale contiene molti campi tra cui:

`next` puntatore all'elemento successivo: tale struttura è infatti più propriamente una lista

`minor` minor number del file di dispositivo corrispondente

`port` puntatore char al registro della porta (A, B, ecc)

`dir` puntatore char al registro che permette di settare la "direzione" della porta, cioè definisce se è un input o un output

`shadow` puntatore char contenente il valore della porta, ossia lo stato dei vari bit;

`dir_shadow` puntatore char al valore contenuto nel registro puntato dal campo dir; si ricorda che i registri `R_PORT_P`x`_DIR` sono write-only e quindi per tenere traccia del valore occorre una variabile di appoggio.

`higalarm`, `lowalarm` sono allarmi che possono essere associati alla specifica porta; sono relativi alle sole porte configurate come input e normalmente sono gestiti in interrupt

Questa struttura viene caricata dalla funzione *open* la quale poi la rende disponibile alle altre funzioni. Per fare ciò la *open* ottiene dal descrittore del file il minor number tramite la macro `MINOR(inode->i_rdev)`. Se il minor number è valido si procede all'allocazione della memoria di una variabile chiamata `priv`, ossia un puntatore alla struttura `gpio_private` e si setta subito il minor number all'interno della struttura. Se la porta è A o B allora la struttura viene compilata appoggiandosi a degli array statici definiti omologamente alla struttura (`port`, `dir`, `shadow`, ecc al plurale).

La funzione open viene chiamata sempre all'apertura del file di dispositivo, ma è chiaro che se tutte le volte che viene aperta la porta viene allocata memoria, questa deve essere rilasciata appena viene rilasciato il descrittore del file di dispositivo (ossia a seguito di una chiamata *close*). Per fare ciò si ricorre alla funzione `gpio_release` e tutte le operazioni eseguite qui dentro vengono fatte "bloccando" l'azione degli interrupt; in altre parole le operazioni compiute sono comprese tra le due chiamate `spin_lock_irq(&gpio_lock)` e `spin_unlock_irq(&gpio_lock)` (vedere capitolo 5 di [4] per maggiori informazioni). Segue la liberazione della memoria dei vari elementi della lista.

Rimangono quindi le due funzioni vere e proprie per l'accesso al dispositivo, ossia *write* e *ioctl*. La prima è implementata tramite `gpio_write` la quale esegue i controlli relativamente al minor number e verifica tramite la macro `access_ok` che il buffer passato dallo user-space sia effettivamente leggibile per tutta la lunghezza `count` (anche questa passata dallo user-space); si procede quindi bloccando l'intervento degli interrupt come in `gpio_release` e andando a settare il valore della porta in funzione di determinate maschere.

Per quanto riguarda `gpio_ioctl` si ha un comportamento analogo in funzione della funzione richiesta. In questo caso l'accesso permette una maggiore interazione; infatti è possibile, oltre che leggere e scrivere il registro relativo alla porta, andare ad accedere ai singoli bit, settare/resettare gli allarmi, ecc.

Il modulo presenta poi la sola funzione di inizializzazione (`gpio_init`) la quale registra il modulo come dispositivo a caratteri (`register_chrdev( GPIO_MAJOR, gpio_name, &gpio_fops)`). Si procede configurando i LED (e quindi le porte) tramite macro del tipo `LED_NETWORK_SET`, `LED_DISK_READ`, ecc in accordo con la configurazione del kernel. Infine vengono settate due funzioni di interrupt tramite `request_irq` come segue:

```
if (request_irq(TIMER0_IRQ_NBR, gpio_poll_timer_interrupt,
                SA_SHIRQ|SA_INTERRUPT,"gpio poll", NULL)) {
    printk(KERN_CRIT "err: timer0 irq for gpio\n");
}
if (request_irq(PA_IRQ_NBR, gpio_pa_interrupt,
                SA_SHIRQ|SA_INTERRUPT,"gpio PA", NULL)){
    printk(KERN_CRIT "err: PA irq for gpio\n");
}
```

La prima *if* setta una funzione (`gpio_poll_timer_interrupt`) che dovrebbe eseguire il controllo ciclico degli allarmi (dovuto al timer) eventualmente scattati chiamando `etrax_gpio_wake_up_check()`. La seconda *if* setta una funzione che che dovrebbe eseguire esattamente la stessa cosa della precedente, ma a fronte di un evento sulla porta A. "Dovrebbe" perchè di fatto ciò non avverà mai in quanto queste funzioni non vengono registrate (o meglio assegnate allo specifico interrupt) a causa di alcune condizioni. Una di queste è il flag `SA_SHIRQ` il quale implica che sia possibile condividere l'interrupt della periferica cosa che non è, soprattuto per il timer; anche se teoricamente si potrebbe pensare di condividere particolari operazioni legate alla temporizzazione di base, ciò potrebbe portare probabilmente ad inaccettabili overhead anche se la funzione è molto rapida (o al limite atomica).

Curiosamente il modulo non ha una funzione di "unload". Questo è probabilmente dovuto al fatto che il sistema non prevede di caricare dinamicamente questo modulo. Se però venisse scorporato dal kernel monolitico sarebbe buona cosa prevedere una funzione di questo tipo, quantomeno per deregistrare il device a caratteri e per "liberarsi" delle funzioni di interrupt (sempre che siano state caricate).

2.3 Access from user space

L'accesso alle porte IO in userspace avviene fondamentalmente in due modi: il primo utilizza il tipico metodo di astrazione dei files in UNIX, ossia l'accesso tramite l'apertura di un file di dispositivo[6] specifico, mentre il secondo permette l'accesso tramite delle chiamate di sistema opportune.

2.3.1 IOCTL

Il primo metodo menzionato permette, come detto, di accedere mediante un file di dispositivo apposito contenuto nella directory `/dev`. Su un file è possibile eseguire tipicamente tutte le operazione di apertura, lettura, scrittura, ecc. In questo caso però non sono state implementate (ragionevolmente) tutte le funzioni; è facile infatti pensare che la funzione *append* sul questo tipo di file di dispositivo è tendenzialmente inutile per il controllo di una porta digitale, e così anche per molte altre chiamate di sistema come per esempio *fseek*. Ciò sarà comunque più chiaro in seguito quando verrà analizzato il driver in modo più dettagliato.

Astrarre una porta digitale tramite un file di dispositivo a caratteri è quindi una cosa possibile, tanto più che le porte in questione (`a` e `b`) sono accessibili indirizzando 8 bit per volta; nonostante ciò il metodo migliore per effettuare il controllo della porta è l'utilizzo della chiamata di sistema *ioctl* la quale permette di eseguire molte operazioni tra cui:

IO_READBITS legge i singoli bit della porta sia in che out. Questa funzione è però deprecata e si consiglia di sostituirla con IO_READ_INBITS e IO_READ_OUTBITS.

IO_SETBITS setta lo stato di uno o più bit, ossia porta il pin corrispondente sul processore a livello logico 0.

IO_CLRBITS resetta lo stato di uno o più bit, ossia porta il pin corrispondente sul processore a livello logico 0.

IO_READDIR permette di leggere la "direzione" del pin, ossia se il pin è in configurazione input o output. Anche questa è deprecata in favore di IO_SETGET_INPUT/OUTPUT.

Di seguito viene riportato un esempio liberamente tratto dal sito della ACMESYSTEM il quale permette di fare lampeggiare il LED rosso presente sulla FOX bard. Come visto in precedenza il LED è connesso sul quarto pin della porta A (ossia su PA3); detto ciò si ha:

[6] `/dev/gpioX` dove `X` sta tipicamente per `a`, `b` o `g`

```
#include "stdio.h"
#include "stdlib.h"
#include "unistd.h"
#include "sys/ioctl.h"
#include "fcntl.h"
#include "asm/etraxgpio.h"
#define DEVICE "/dev/gpioa"
int main(void) {
     int fd, i;
     int iomask;
     if ((fd = open(DEVICE, O_RDWR))<0) {
            printf("Open error on %s\n",DEVICE);
            exit(-1);
     }
     iomask=1<<3; //maschera che indica il 4° bit, ossia PA3
     printf("Blinking LED %i\n", iomask);
     for (i=0;i<20;i++)      {
            printf("Led ON\n");
            ioctl(fd,_IO(ETRAXGPIO_IOCTYPE,IO_SETBITS),iomask);
            sleep(1);
            printf("Led OFF\n");
            ioctl(fd,_IO(ETRAXGPIO_IOCTYPE,IO_CLRBITS),iomask);
            sleep(1);
     }
     close(fd);
     return 0;
}
```

Questo programma non fa altro che accendere e spegnere 20 volte il led rosso.

2.3.2 Syscall

Il secondo metodo permette di eseguire il controllo della porta specificata tramite delle chiamate di sistema (*syscall* appunto) dirette. Queste chiamate sono definite in `linux/gpio_syscalls.h` e permettono di accedere direttamente al kernel senza dover ricorrere ad un descrittore, quindi non c'è più bisogno di aprire un file di dispositivo e quindi senza dover attuare l'apertura del file per poter eseguire le funzioni di sistema associate al file. Si osservi che una chiamata di sistema relativa al file di dispositivo (come appunto *open* e *ioctl*) fa passare il processo da user-space a kernel-space e questa richiesta al kernel viene fatta tramite un interrupt software[7] il quale normalmente richiede molti cicli soprattutto per salvare il contesto del processo; tipicamente questi eventi degradano molto le performance del programma se sono ripetute con una certa frequenza. Le syscall implementate nel file `gpio_syscalls.h` sono nettamente più prestanti perchè non hanno i problemi appena citati.

Un esempio simile al precedente è il seguente:

```
#include "stdio.h"
#include "stdlib.h"
#include "unistd.h"
#include "sys/ioctl.h"
#include "fcntl.h"
#include "time.h"
#include "string.h"
#include "linux/gpio_syscalls.h"

int main(void) {
   int i;
   gpiosetdir(PORTA, DIROUT, PA3); //set PA0 as output
   for(i=0; i<10; i++)
    {
      gpiosetbits(PORTA, PA3); //PA3 è il led ROSSO
      printf("%d\n", (gpiogetbits(PORTA, PA3))?(1):(0));
      sleep(1);
      gpioclearbits(PORTA, PA3);
      printf("%d\n", (gpiogetbits(PORTA, PA3))?(1):(0));
      sleep(1);
```

[7] Tipicamente tramite l'interrupt 80 con l'istruzione `int 0x80` nei sistemi x86.

```
    }
   return(0);
}
```

2.4 Access form kernel space

A livello kernel l'accesso alle porte GPIO è relativamente semplice e non è diverso da ciò che si esegue quando si programma un comune micro-controllore con periferiche memory mapped. Anche in questo caso infatti non si deve fare altro che scrivere ad un opportuno indirizzo di memoria (registro) uno specifico valore.

Considerando nuovamente la porta `A`, si ottiene dal datasheet [1] che il registro di configurazione di tale porta è collocato all'indirizzo `0xB0000030`. Questo registro è a 32 bit, come è ovvio che sia visto che questa è la "word" del processore ETRAX 100LX. Sempre da [1] si ha che il nome di default di questo registro è `R_PORT_PA_SET`, ossia lo stesso che si può trovare nella configurazione del kernel. Esaminandolo si osserva che pur essendo a 32 bit, i bit effettivamente destinati alla porta PA sono solo i primi 16: i primi 8 bit identificano lo stato dei singoli pin, mentre gli 8 bit più significativi (dei 16 bit meno significativi in esame) indicano al processore se il bit corrispondente è un input o un output. Questi due registri si chiamano `R_PORT_PA_DATA`, `R_PORT_PA_DIR` rispettivamente pari a `0xB0000030` e `0xB0000031`. Il registro `R_PORT_PA_DIR` è inoltre write-only.

Gli esempi che seguono verificano quanto detto. Si consiglia a chi non avesse esperienza in termini di programmazione del kernel di leggere [4]. Per quanto riguarda la compilazione dei moduli si rimanda all'appendice A.

2.4.1 Retrieve Informations

Il modulo proposto non fa altro che leggere l'indirizzo e il contenuto dei registri relativi alla porta PA:

```
/** INFOPA.c **/
#include <linux/module.h>
#include <asm/io.h> //provide access to GPIO port and other
MODULE_LICENSE("DUAL GPL/BSD");
MODULE_VERSION("0.1");

/** Funzione di scaricamento del modulo **/
void infoPA_cleanup_module(void) {
//Non devo fare nulla
}
/** Funzione di inizializzazione del modulo **/
int infoPA_init_module(void) {
//Il cast a int è per evitare warning in fase di compilazione
   printk(KERN_ALERT "\nIndirizzi della porta PA:\n"
          " -R_PORT_PA_SET  = %8X\n -R_PORT_PA_DATA = %8X\n"
          " -R_PORT_PA_DIR  = %8X\n", (int)R_PORT_PA_SET,
          (int)R_PORT_PA_DATA, (int)R_PORT_PA_DIR);

   printk(KERN_ALERT "\nValori della porta PA:\n"
          " -R_PORT_PA_SET  = %8X\n -R_PORT_PA_DATA = %8X\n"
          " -R_PORT_PA_DIR  = %8X\n", *R_PORT_PA_SET,
          *R_PORT_PA_DATA, *R_PORT_PA_DIR);

   return 0;
}
/***************************************
 * INIT & EXIT
 ***************************************/
module_init(infoPA_init_module);
module_exit(infoPA_cleanup_module);
```

Il modulo scritto è molto semplice ed è costituito dalle sole due funzioni fondamentali di inizializzazione e cleanup; la prima non fa altro che scrivere nel ring buffer del kernel gli indirizzi e i contenuti dei registri in questione. In fase di unload del modulo non viene eseguito nulla di specifico in quanto non sono state fatte operazioni di allocazione della memoria, definizioni di directory in `/proc`, sostituzioni di syscall, ecc. Per verificare il codice è sufficiente collegarsi alla scheda, caricare il modulo e vedere le ultime righe scritte nel file `/var/log/message`:

```
telnet 192.168.0.90         #tipica modalità per connettersi alla FOX
insmod /mnt/flash/infoPA.ko #carico il modulo copiato in /mnt/flash
tail /var/log/message       #leggo il report del kernel
rmmod infoPA                #eventualmente tolgo il driver dal kernel
```

Nota: Per fare in modo che il kernel scriva in `/var/log/message` si prepone alla stringa della formattazione della funzione printk la macro `KERN_ALERT`. Di queste macro ne esistono altre che permettono la gestione dei vari messaggi in funzione della gravità del messaggio; una di queste altre macro è `KERN_INFO` e anche questa macro fa si che il messaggio venga salvato in `/var/log/message`. Ciò è differente da quello che accade nei sistemi tradizionali dove tipicamente con `KERN_INFO` i log vengono indirizzati in `/var/log/message`, mentre con `KERN_ALERT` i log vengono redirezionati in `/var/log/syslog`[8].

2.4.2 Change status of LEDs

Il modulo seguente è un po' più complesso. Ora si vuole fare in modo di istruite il kernel della FOX per cambiare lo stato dei due led rosso e giallo. Per fare ciò l'idea proposta è quella di usufruire del filesystem virtuale `/proc`. Verrà quindi creato un file nella sottodirectory `calzo/` chiamato `pa` scrivendo il quale si ottiene il cambiamento di stato dei pin configurati come output. Verranno quindi spiegate mano a mano tutte le funzioni del modulo.

"La completezza è nemica della chiarezza": per questo i moduli presentati sono volutamente incompleti e possono presentare qualche baco, ma dovrebbero essere più semplici da capire.

Dichiarazioni e Macro

```
/** CALZOLED.C **/
#include <linux/module.h>
#include <linux/proc_fs.h> //proc interface
#include <asm/uaccess.h>   //copy_from_user, copy_to_user, ecc
#include <asm/semaphore.h> //semaphore structure
#include <asm/io.h>        //access to GPIO port and other
                           // peripherals on ETRAX LX100
MODULE_LICENSE("DUAL GPL/BSD");
MODULE_VERSION("0.1");

/** Parametri del modulo **/
static unsigned int setdataPA = 0x00;
module_param(setdataPA, uint, S_IRUGO|S_IWUSR); //[4] cap 2, pag36
MODULE_PARM_DESC(setdataPA, "Set PA port");
struct semaphore sem; //mutual exclusion semaphore - x evitare le
                      //"Concurrency and Race conditions" [4] cap5
//Strutture relative a /proc filesystem
struct proc_dir_entry *proc_cldir = NULL; //main directory
struct proc_dir_entry *proc_clpa  = NULL; //file per la porta PA
/** Strutture relative al filesystem proc **/
#define PROC_CL_DIR "calzo"  //directory in /proc
#define PROC_CL_PA  "pa"     //file in /proc/calzo
...
/****************************************
 * INIT & EXIT
 ****************************************/
module_init(cl_init_module);
module_exit(cl_cleanup_module);
```

Le due macro `module_init` e `module_exit` sono implementate alla fine del file.

Inizializzazione del modulo

```
int cl_init_module(void) {
   //Inizializzazione dei semafori
   init_MUTEX(&sem);
   //Creazione della directory virtuale /calzo in /proc
```

[8]Questo è almeno ciò che avviene in Slackware; in altre distribuzioni i nomi possono cambiare

```
    proc_cldir = proc_mkdir(PROC_CL_DIR, NULL);

    if(!proc_cldir)
      {
        printk(KERN_ALERT "Unable to create proc_cldir\n");
        return -ENOMEM;
      } else
      {
        proc_clpa = create_proc_read_entry(
                      PROC_CL_PA, //nome del file
                      0, //protection mask: default=0
                      proc_cldir, //parent dir
                      NULL, //funzione di lettura del file
                      NULL);
        if(!proc_clpa)
          {
            printk(KERN_ALERT "Unable to create proc_clpa\n");
            cl_cleanup_module();
            return -ENOMEM;
          } else
            proc_clpa->write_proc = cl_write_pa;
       }
     printk(KERN_ALERT "calzoled caricato.\n"
           " PA data:      %x\n PA direction: %x\n",
           *R_PORT_PA_DATA, *R_PORT_PA_DIR);
     return 0;
}
```

La funzione non fa altro che inizializzare i semafori usati più avanti nella funzione *write*. Successivamente viene creata la directory `calzo/` in `/proc` con la chiamata `proc_mkdir`. Se non vi sono problemi si procede alla creazione del file virtuale (con `create_proc_read_entry`) `pa`. Prima di chiudere la funzione, vengono stampati i valori della porta PA nel ring buffer del kenrel.

Clean-up del modulo

```
void cl_cleanup_module(void) {
   // ELIMINAZIONE delle entità in proc
   if(proc_clpa)
     remove_proc_entry(PROC_CL_PA, proc_cldir); //file e parent-dir
   if(proc_cldir)
     remove_proc_entry(PROC_CL_DIR, NULL);

   printk(KERN_ALERT "calzoled scaricato\n");
}
```

Eliminazione dei file virtuali in `/proc` tramite la chiamata `remove_proc_entry`.

Funzione write per il file virtuale `pa`

```
int cl_write_pa(struct file *file, const char __user *buffer,
                unsigned long count, void *data) {

   unsigned char buf=0;
   int new_value=0; //se il nuovo valore è negativo c'è un errore

   if(!count)      //se il contatore è 0...
     return count; //...ritorna 0 e non fare nulla
   if(!buffer)     //se il buffer non esiste...
     return 0;     //...0, ma sarebbe meglio restituire un errore
   if(down_interruptible(&sem)) //evitare le concurrency conditions
     return -ERESTARTSYS;

//Se sono qui, procedo con la copia del buffer passato
// dall'user-space per un massimo della linghezza del
// buffer di destinazione.
   copy_from_user((void*)&buf, buffer, 1);
```

```
//converto il carattere ESADECIMALE in un numero
   if ( buf>='0' && buf<='9')
     new_value += ((int)buf-'0');
   else if ( buf>='A' && buf<='F')
     new_value += ((int)buf-'A'+10);
   else if ( buf>='a' && buf<='f')
     new_value += ((int)buf-'a'+10);
   else
     new_value = -1;

// Setto i dati della porta A se non ci sono errori
   if(new_value<0)
     printk(KERN_ALERT "Warning: %c is not HEX format.\n", buf);
   else
     *R_PORT_PA_DATA = new_value;

   up(&sem);     //rilascio il semaforo
   return count; //ritornando count, si fa tutto in una sessione
}
```

La funzione *write* per il file system *proc* in esame ha una struttura di parametri relativamente diversa da quella della funzione *write* tradizionale, ma il concetto è comunque lo stesso. Dopo i controlli di rito iniziali viene controllato il primo carattere di un buffer passato dallo spazio utente e lo trasforma in un numero se questo carattere è rappresentativo di un numero esadecimale. É chiaro che analizzando solo una lettera, i valori plausibili che verranno scritti nel registro della porta A andranno da 0 a 15, ossia verranno fatti variare solo i primi quattro bit. Ciò è più che sufficiente in quanto i LED della FOX sono connessi proprio sui primi quattro pin relativi alla porta in questione.

Utilizzare il modulo

Una volta compilato il modulo e caricato sulla FOX (per esempio in `/mnt/flash`) occorre caricato con il solito comando *insmod*. Una volta fatto questo si troverà nel filesystem `/proc` la directory `calzo/` e il file `pa`. Ora con il comando

```
echo F > /proc/calzo/pa
```

i primi quattro pin della porta PA verranno portati allo stato logico alto se configurati come output. I LED controllati in logica inversa si spegneranno. Viceversa scrivendo `0` al posto di `F`, i LED si accenderanno.

2.5 Interrupt

Si vuole ora provare a fare in modo che venga attivata una funzione di interrupt a fronte di un evento sulla porta A. Nella sezione 2.2 è stata introdotta la sintassi tramite la quale è possibile "agganciare" una funzione di interrupt all'evento specifico. Il codice precedentemente descritto era:

```
request_irq(PA_IRQ_NBR, gpio_pa_interrupt,
            SA_SHIRQ|SA_INTERRUPT, "gpio PA", NULL)
```

ma come detto questa funzione non viene mai registrata se lanciata con il flag `SA_SHIRQ`; se tolto, la funzione `gpio_pa_interrupt` viene registrata con successo come si può vedere da `cat /proc/interrupt` nella figura 3. A questo punto per debaggare il sistema conviene scrivere per esempio `printk(KERN_ALERT "interrupt chiamato!\n")` all'inizio di `gpio_pa_interrupt` in modo da vedere se l'interrupt viene scatenato oppure no. Se ora si preme il pulsante della FOX è presumibile pensare che venga generato un evento e quindi "`interrupt chiamato!`" dovrebbe comparire in `/var/log/message`, ma così non è. Occorre infatti fare un'altra cosa che non è fatta nel kernel, ossia occorre agire opportunamente sul registro `R_IRQ_MASK1_SET` che indica su quali pin della porta A può essere scatenato l'interrupt. A tal proposito si può modificare la funzione di inizializzazione del modulo *infoPA* descritto in precedenza (o se si preferisce, si scrive un altro modulo) aggiungendo la riga `*R_IRQ_MASK1_SET=0xF` tramite la quale viene settato che l'interrupt scatterà sui primi quattro pin della porta A se configurati come input; ciò implica che l'unico pin che può generare un evento è quello legato al pulsante.

A questo punto però sorge un problema: si supponga di connettersi e di caricare il modulo che permette di abilitare effettivamente gli interrupt. A questo punto, senza che si tocchi nulla, si noterebbe un comportamento alterato della scheda la quale si trova improvvisamente a dover soddisfare una enorme coda di interrupt! Premendo il pulsante si noterebbe inoltre che la scheda riuscirebbe a liberare parte della coda, ma dopo alcune pressioni si avrebbe da parte del kernel un messaggio del tipo:

```
axis kernel: Disabling IRQ #11
```

ossia verrebbe disabilitato l'interrupt. A questo punto (se non ancora prima) conviene riavviare la FOX.

Questo comportamento è tutto sommato corretto perchè da [2] si legge che gli interrupt sul processore ETRAX 100LX relativi alla porta PA sono "a livello" alto, ossia l'interrupt scatta se il livello del pin di ingresso rimane al livello alto. Il pulsante è connesso a logica inversa direttamente al microprocessore il quale, ovviamente, allo stato di riposo rileva il livello logico alto e quindi una serie interminabile di interrupt.

Per testare gli interrupt verrà quindi scritto un nuovo modulo chiamato `irqPA.c`.

2.5.1 irqPA

L'obiettivo di questo modulo è fare in modo che l'interrupt su un pin della porta PA venga sì rilevato e servito, ma evitando i problemi appena menzionati. Perfare ciò vi sono molte possibilità e l'idea scelta è quella di rendere input un pin non connesso a nulla e di comandarlo tramite un altro pin (di output) in userspace. Esaminando la struttura della scheda relativa alle GPIO (vedere [5]) si ha che la scelta migliore è quella di convertire il pin PA0 come input, mentre , come pin per il controllo, si è scelto casualmente il pin PA3 (led rosso)[9]. Per connettere i due pin è meglio ricorrere a una resistenza dell'ordine di $1K\Omega$ come mostrato in figura 4.

Dal punto di vista software, per fare quanto richiesto occorre prima di tutto configurare il kernel in modo che i primi due pin della porta A vengano configurati come output. Per fare questo si lanci la configurazione del kernel e si modifichi la voce `R_PORT_PA_DIR` (figura 2) al valore `0x1C`. Ovviamente il kernel andrà ricompilato e riscritto sulla FOX. Per vedere se la modifica è andata a buon fine si lanci il comando *readbits* il quale deve restituire come primi caratteri (relativi alla porta A) "`111XXX10`".

Il modulo che verrà presentato deve eseguire queste operazioni:

loading in fase di caricamento deve registrare con successo funzione di interrupt

interrupt in fase di interrupt deve servire la funzione, ma al tempo stesso portare allo stato logico basso il pin PA3; ciò è obbligatorio per evitare, come gia detto, che il sistema venga subissato di richieste di interrupt. Questo comportamento è strettamente elegato a questo esempio e non è la norma.

unloading in fase di clean-up del modulo occorre necessariamente deallocare la funzione di interrupt

[9] Si noti che questa è già una configurazione di default.

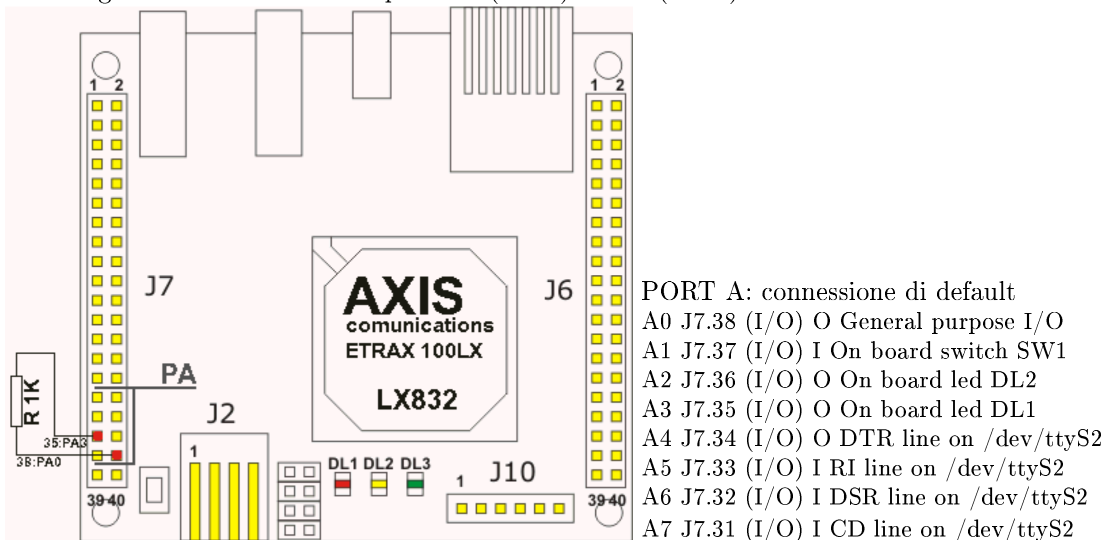
Figura 3: Output di `cat /proc/interrupt`.

```
       CPU0
 2:   25498   CRISv10  timer
 3:       0   CRISv10  fast timer int
 6:       0   CRISv10  ETRAX 100LX built-in ethernet controller
 8:       0   CRISv10  serial
11:       0   CRISv10  gpio PA
16:      40   CRISv10  ETRAX 100LX built-in ethernet controller
17:     112   CRISv10  ETRAX 100LX built-in ethernet controller
22:     156   CRISv10  serial 0 dma tr
23:       0   CRISv10  serial 0 dma rec
24:       0   CRISv10  ETRAX 100LX built-in USB (Tx)
25:       0   CRISv10  ETRAX 100LX built-in USB (Rx)
31:       2   CRISv10  ETRAX 100LX built-in USB (HC)
```

Si osservi che il massino numero di interrupt ammesso dal processore ETRAX 100LX è 32; il limite è settato nel software mediante la macro "`#define NR_IRQS 32`" in `$AXIS_KERNEL_DIR/include/asm-cris/arch-v10/irq.h`.

```
#include <linux/interrupt.h>
#include <linux/module.h>
#include <asm/io.h> //provide access to GPIO port and other

MODULE_LICENSE("DUAL GPL/BSD");
MODULE_VERSION("0.1");

#define IRQ_PA_MASK 0x01 //interrupt sul pulsante

static DEFINE_SPINLOCK(gpio_lock_irq);

/***** FUNZIONE DI INTERRUPT ******
* In questa funzione viene disabilitata temporaneamente la
* possibilità di ricevere altri interrupt fino a che non è
* soddisfatta la funzione seguente
* *******************************/
static irqreturn_t
  irqPA_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
   *R_IRQ_MASK1_CLR = 0;
   *R_PORT_PA_DATA  = 0; //fondamentale eliminare la
                         // causa dell'interrupt!
   printk(KERN_ALERT "calzo - irq servito\n");
   return IRQ_HANDLED;
}
/****************************************
* INITIALIZATION & CLEANUP functions
****************************************/
/// Funzione di inizializzazione del modulo
int irqPA_init_module(void) {
   if (request_irq(PA_IRQ_NBR, irqPA_interrupt,
                   SA_INTERRUPT,"calzo gpio PA interrupt", NULL))
     {
        printk(KERN_CRIT "err: PA irq for gpio (calzo)\n");
        return -ERESTART;
     }
   //setto su quale pin vi è l'interrupt valido
   *R_IRQ_MASK1_SET = IRQ_PA_MASK;
   return 0;
}

///Funzione di scaricamento del modulo
void irqPA_cleanup_module(void) {
   spin_lock_irq(&gpio_lock_irq);
```

Figura 4: Connessione dei pin PA0 (J7.38) e PA3 (J7.35) su FOX board con resistenza da $1K\Omega$



```
    *R_IRQ_MASK1_SET = 0;
    *R_IRQ_MASK1_CLR = 0;
    free_irq(PA_IRQ_NBR, NULL);
    spin_unlock_irq(&gpio_lock_irq);
}

/**************************************
* INIT & EXIT
**************************************/
module_init(irqPA_init_module);
module_exit(irqPA_cleanup_module);
```

Nota: nella funzione `irqPA_cleanup_module` viene usata la funzione `spin_lock_irq(&gpio_lock_irq)` la quale disabilita temporaneamente gli interrupt sul processore corrente. Ciò forse non è correttissimo o comunque potrebbe essere inutile visto che comunque si agisce su `R_IRQ_MASK1_SET` andandolo a resettare, disabilitando quindi gli interrupt relativi a PA.

Si ricorda che per usare questo modulo è opportuno commentare la registrazione della funzione di interrupt nel file `$AXIS_KERNEL_DIR/arch/cris/arch-v10/drivers/gpio.c` in modo da essere sicuri che non vi siano altre funzioni già registrate.

Utilizzare il modulo

Una volta connessi alla FOX e caricato il modulo con il comando *insmod*, se non viene segnalato alcun errore, si può controllare che il modulo sia correttamente caricato eseguendo `cat /proc/interrupt` che deve restituire un messaggio identico se non uguale a quanto riportato in figura 3.

A questo punto se non vi sono problemi occorre portare a "1" la porta PA3 con il comando "`setbits -p a -b 3 -s 1`". Così facendo il pin PA0 viene portato al livello alto e l'interrupt viene rilevato e quindi servito. Ora si potrebbe pensare di vedere il LED rosso spegnersi, ma così non è perchè al massimo dopo poche decine di microsecondi la funzione di interrupt viene servita; questa procede all'azzeramento di `*R_PORT_PA_DATA`, operazione che è obbligatoria. Chiaramente sarebbe più corretto resettare il solo pin che genera l'interrupt (PA0), ma per chiarezza e semplicità si è preferito procedere in questo modo.

Appendici

A How to configure&compile kernel module

Quanto verrà qui esposto è relativo al kernel della FOX board, ma è tendenzialmente applicabile ad ogni altro kernel.

A.1 Configuring kernel modules

Potrebbe essere necessario o quantomeno utile inserire nell'interfaccia di configurazione del kernel la possibilità di scegliere i moduli scritti e/o di poterli configurare. Per farlo occorre creare un file chiamato `Kconfig` nella directory che contiene i nuovi file che si vogliono compilare. È chiaro che se i files sono in una directory già presente nel kernel, sarà sufficiente editare il file `Kconfig` che si troverà già in quella directory. In questo esempio si considera sempre la directory `$AXIS_KERNEL_DIR/drivers/calzo`.

Per prima cosa si esaminerà `Kconfig` contenuto della sottodirectory `arch/cris/` del kernel il quale compone i vari sottomenù della root del sistema di configurazione, ossia la prima videata che appare lanciando make menuconfig. Aggiungendo in fondo al file:

```
menu "Calzo Device"
source "drivers/calzo/Kconfig"
endmenu
```

ossia aggiunge la voce *Calzo Device* all'interfaccia di configurazione la quale è un vero e proprio sottomenù. Ora occorre creare il file `drivers/calzo/Kconfig` (anche vuoto) per evitare il fallimento della creazione dell'interfaccia di configurazione. Aggiungendo al file le seguenti voci:

```
config CALZO_MODULES_ENABLE
       bool "Attivare Calzo Modules"
       help
         Compariranno alcuni moduli di esempio scritti da Calzo per
         Linux Day 2007 come esempio

config CALZO_INFOPA
       tristate "infoPA"
       depends on CALZO_MODULES_ENABLE
       default m
       help
         infoPA è un modulo che, una volta caricato, restituisce
         gli indirizzi e il contenuto della porta A su ETRAX LX100
         Compilato come modulo, il nome è infoPA.ko

config CALZO_LED_PA
       tristate "Calzo LED"
       depends on CALZO_MODULES_ENABLE
       default m
       help
         Permette di settare lo stato dei LED sulla porta A
         scrivendo un carattere esadecimale nel file /proc/calzo/cl.
         Per esempio echo F > /proc/calzo/cl spegne tutti i LED.
         Compilato come modulo, il nome è calzoled.ko
```

A questo punto avviando la configurazione del kernel, attivando queste nuove opzioni e salvando, verrà creato un nuovo file `.config` contenente per esempio:

```
CONFIG_CALZO_MODULES_ENABLE=y
CONFIG_CALZO_INFOPA=m
CONFIG_CALZO_LED_PA=m
```

Come noto verrà creato poi un file .h contenente le stesse informazioni in termini di definizione, ma ancora prima queste macro verranno usate per istruire il compilatore sui files da compilare.

A.2 Compiling kernel module

Per compilare un modulo del kernel si consiglia di creare una directory nella root dei sorgenti del kernel della FOX. Nel caso in esame, supponendo che i sorgenti dell'intero sistema siano in `/usr/local/fox`, ci si sposta nella directory `os/linux-2.6` ossia nella directory dei sorgenti del kernel. Una volta qui si può per esempio creare una cartella `drivers/calzo/` con il comando `mkdir drivers/calzo/`; a questo punto editare il `Makefile` contento nella directory `drivers/` aggiungendo (per esempio in fondo) "`obj-y += calzo/`".

Così facendo il sistema di compilazione del kernel saprà che dovrà cercare un `Makefile` all'interno della directory `derivers/calzo/` per questo occorrerà creare un file `Makefile` anche vuoto, altrimenti la compilazione del kernel fallirebbe. All'interno di questo makefile occorre indicare quali (nuovi) moduli compilare. Supponiamo di avere due moduli chiamati `calzoled.c` e `infoPA.c` e di scrivere:

```
obj-y += calzoled.o
obj-m += infoPA.o
```

Queste istruzioni faranno si che `calzoled.c` sia compilato e inglobato direttamente nel kernel, mentre `infoPA.c` verrà compilato come modulo e quindi nella directory di appartenenza comparirà `infoPA.ko`; ovviamente l'estensione `.ko` è legata al fatto che si stà considerando la versione 2.6 del kernel.

Nella sezione A.1 si è visto come configurare un modulo. Per fare in modo che il compilatore sia legato alla configurazione da noi impostata è possibile scrivere:

```
obj-$(CONFIG_CALZO_LED_PA) += calzoled.o
obj-$(CONFIG_CALZO_INFOPA) += infoPA.o
```

dove le due macro hanno valore `y` o `m` in funzione che si voglia includere il modulo direttamente nel kernel o lo si voglia tenere al di fuori per caricarlo in seguito.

Per compilare il modulo in modo non monolitico è sufficiente lanciare `make` nella directory del kernel. Si ricorda che per cross-compilare una qualsiasi applicazione per FOX occorre aver eseguito il comando init_env nella directory principale come spiegato in [5]:

```
cd /directory/della/fox/devboard-R2_01
. init_env
```

Questo ultimo comando creerà le variabili di ambiente necessarie per la compilazione con compilatore *cris*. Le variabili diventano per esempio:

```
AXIS_TOP_DIR = /directory/della/fox/devboard-R2_01
AXIS_KERNEL_DIR = /directory/della/fox/devboard-R2_01/os/linux-2.6
```

Per compilare è poi sufficiente lanciare `make` o `make modules` se le modifiche vanno solo a creare dei moduli. Se i moduli sono stati compilati e inglobati nel codice del kernel occorre riflashare la fox, mentre se i moduli sono a se stanti (ossia esiste un file .ko) li si può copiare come un normale programma con il comando `scp`; tipicamente:

```
scp nome_modulo.ko root@192.168.0.90:/mnt/flash
```

Si ricorda che `/mnt/flash` è una directory sicuramente scrivibile internamente alla flash.

Riferimenti

[1] Datasheet ETRAX 100LX

[2] ETRAX 100LX Designer Manual - etrax_100lx_des_ref-060209.pdf

[3] ETRAX 100LX Programmer Manual - etrax_100lx_prog_man-050519.pdf

[4] Linux Device Driver 3rd edition

[5] http://www.acmesystems.it/?id=711 come cross-compilare un programma

Info&Credits

Si ringrazia vivamente **MCM Energy Lab** e la sezione di Azionamenti del dipartimento di **Ingegneria Elettrica** del Politecnico di Milano per aver dato pieno supporto e l'hardware per effettuare i test.

Il presente documento è stato scritto inizialemnte per la 7° giornata di Linux e del Software libero e presentato per Mantova dal **LUGMan** (`www.lugman.org`) a Sangiorgio ed è rilacsiato sotto licenza GPL v2. Chiunque volesse avere i sorgenti del documento può richiederlo a info@lugman.net o consultare il sito dell'associazione. Il documento è stato scritto con LyX 1.4.3 sotto GNU/Linux Slackware 10.2. Tutto il software riportato è stato scritto e testato (cross-compilato) con GNU/Linux Slackware 10.2.

Writer: *Calzo* (calzog @ gmail . com)

release 1 October 2007 for Linux DAY 2007 in Mantova (Italy) - scritto da Calzoni Pietro aka *Calzo*, membro del LUGMan, Linux Users Group Mantova

selease 2 March 2008 - correzioni minori e traduzione in inglese by *Calzo*. Grazie a tutti coloro che hanno mostrato interesse per questo documento.

release 3 August 2008 - correzioni minori. Grazie in particolare a Geert Vancompernolle par i feedback soprattutto sulla versione inglese.