FOX DOC

28 October 2007
Linux Day

Abstract

Goal of this document is try to incline who read to the Embedded Linux world. The object of this article is the FOX Board because it is a product complete by the point of view of hardware as much as software and moreover it is a tools versatile, simply to use and completely free.

Mainly this article summarizes the experience about this board made by the undersigned (and clearly by who wish extend the document). The arguments can go from hardware to software user/kernel space.

Actually the focus is oriented on GPIO device by which it is possible to control the LED (included LEDs of ethernet connector) and the button.

1 Introduction to the ETRAX 100LX

Before begining it is better to give some informations about the processor. This informations are available by the datasheet and obviously they are more detailed in there. Now it is enough to known the ETRAX 100LX is a RISC 32-bit processor (100MIPS[1]) with 8kbyte of internal cache. Every peripherals are memory-mapped and this imply that exists at least one particular physical memory address (named register) for each peripheral in which every bit set a specific function or behavior of the considered device. This is very convenient because allow to menage peripherals directly and in a simple manner without use particular instructions. Examples of processor no memory-mapped are those of *x86* family.

Memory Total virtual addressing is 4Gb and the access to the external memory is menaged by integrated controller on chip; this controller get the interface to the memory SDRAM, EDO SRAM, EPROM, EEPROM and Flash PROM without using external device or transceiver. There is also a MMU unit.

I/O 2 synchronous and 4 asynchronous serial port, 2 parallel port IEEE1284 compliant, SCSI-2 and SCSI-3, IDE/ATA-2, 2 USB (both host and device!), Ethernet 10/100 fullduplex.

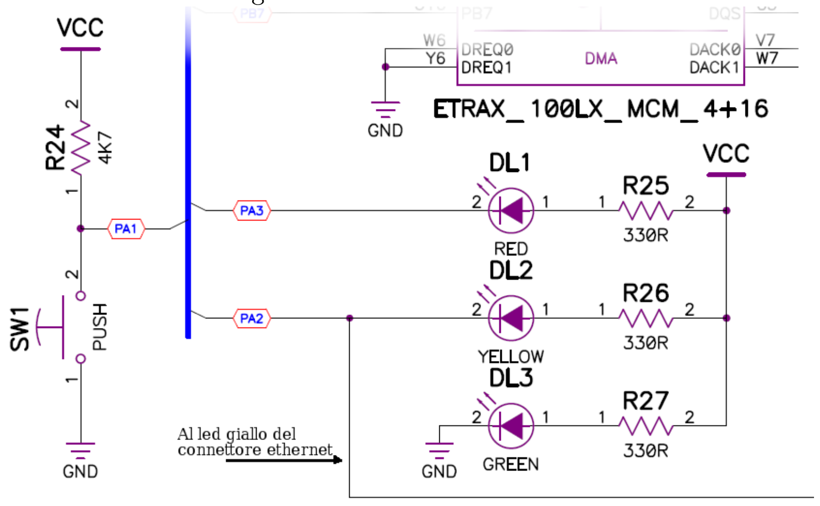
[1] Mega Instruction Per Second

2 GPIO

GPIO indicates *General Purpose Input/Output*, that is a device of input or output for general purposes. Those kind of devices called "port" are mainly two indicated with A and B[2] both composed by 8 bit. To this port correspond 8 pin on socket processor and their state and configuration type are indicated by others 8 bit in specific registers. These port are digital and is called tri-state because operating on a particular 8-bit register (`R_PORT_PA_DIR` for A port), it is possible to configure a single pin as an input (0) or output (1). If it is an output, manipulating an other register (`R_PORT_PA_SET` for A port) will possible set in high (1) or low (0) state every pin.

In figure 1 is reported the connection schematic of some port on FOX board. As we see there are three LED, one of these always on (green) to indicate power is present, while the other two (yellow and red) are connected directly to the pin 3 (PA2) and 4 (PA3) of micro-processor. The button (switch SW1) is otherwise connected to the port 2 (PA1).

Figura 1: Button and LED connections on fox board.
This image is taken from board schematics.



As you notice LEDs are connected in negate logic, so anode of diode is connected (with a resistor) to +Vcc, while cathode is connected directly on processor pin. So we can turn on the led by setting 0 (low logical state) to the corresponding pin of port which LED is connected. Similarly the switch drive down (low logic state) the pin PA1 when will be pressed, while on rest state (open circuit) pin will set on high state. This choice is made because when the diode is on, the current through in it and enter in the corresponding pin of the microprocessor; but this pin is in low state, so the voltage between it and

[2] There would be also a thirth port called G, but it is not mentioned in this article

ground is about 0V. Then the power dissipated from the pin of processor is

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

and for this reason it is possible to provide a current also near to the nominal current for each pin[3] and however the processor will be less hot. If the diodes would connected in direct mode (alias direct logic) and if the current of the pin of processor would be 12mA, the power dissipated would be

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

Unfortunately this circuit has a problem: the button drive directly to the ground the potential of the pin which it is connected. Now if the pin would be configured as an output and its state would be set to high (+5V), when it will be pressed a short-circuit happening. This normally cause a permanently break down of the pin or whole processor. It would be a good thing connect a resistor between switch and pin PA1.

2.1 Kernel configuration

Default configuration of port A and B is done at kernel level. Considering the kernel 2.6 and moving in subdirectory `os/linux-2.6/` we can execute the typical command `make menuconfig`; if otherwise we prefer to control all from main directory (typically `devboard-R2_01`) we must type `make kernelconfig`. Any way standard textual screen for kernel configuration will appear.

To configuring the "specific" device of this processor like GPIO port we must move in section **Hardware setup** as we can see in figure 2. It is possible to set four value by which we can indicate how LED are connected to the port A or B; but this last one have some shared peripherals and the other functionalities (like SCSI or USB) are priority then genetic I/O function. With `CSP0` (Cable Selected Port) we can enable the special function of port B specifying if the functionality of every pin (from 2 to 7) is "special" or not (simple I/O). Choosing one of the first three option, other voice which allow to set to which pin LEDs are connected will be appear.

The last function (*None*, showed in figure 2) set only the option `R_PORT_PA_DIR`, `R_PORT_PA_DATA`, `R_PORT_PB_CONFIG`, `R_PORT_PB_DIR`, `R_PORT_PB_DATA` indicated just below. These options define the exact value to put in the specific register; they are present also in the three previous options, but they don't have effect in the previous case if one of the first options is being choosen. Consider now the port A[4]: by default is indicated `R_PORT_PA_DIR=0x1D` and `R_PORT_PA_DATA=0xF0`. Examining the bit value, we see that `0x1D=0b00011101` e `0xF0=0b11110000` so the port PA1 will be configured as input[5], while PA0, PA2 (yellow LED), PA3 (red

[3]Nominal current for each pin is 12mA, but it is better the current would be the smaller as possible by avoid processor over temperature. It is also better to know the maximum power dissipable by the processor.

[4] In Fox 8+32 considered now, LEDs are connected to the port `A`.

[5]As indicated, by default is indicated pin 2 as input to avoid problems mentioned in paragraph 2.

LED) and PA4 are output. These last LED are all on because the first four bits of `R_PORT_PA_DATA` are 0.

2.2 Device driver

The driver in exam is `$AXIS_KERNEL_DIR/arch/cris/arch-v10/drivers/gpio.c` and it menages all the GPIO ports. The major number of this module is assigned to 120 defined by macro

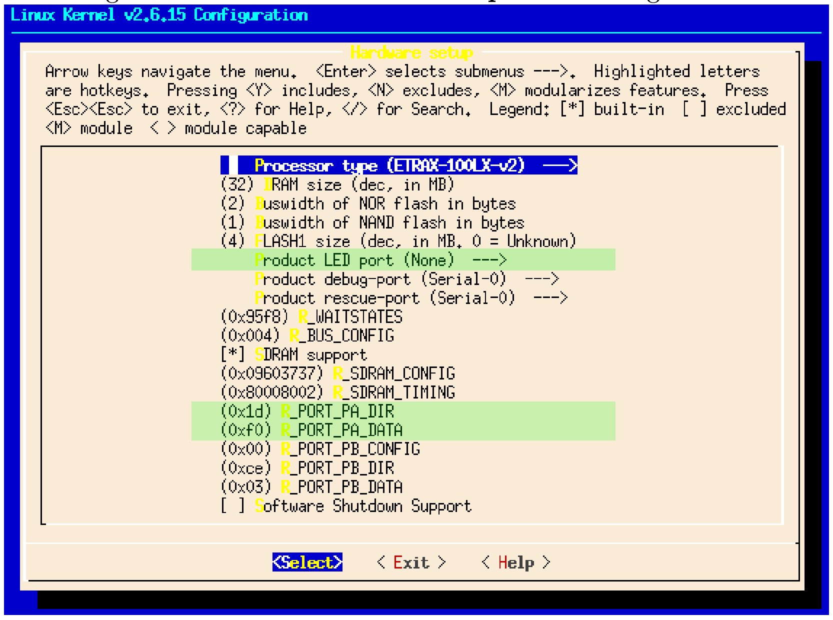
```
#define GPIO_MAJOR 120
```

with minor number 0 and 1 for ports A and B. The module implements the following I/O function for the device:

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

which are assigned to the file operation structure:

Figura 2: Section **Hardware setup** kernel configuration.



```
Linux Kernel v2.6.15 Configuration

                              Hardware setup
 Arrow keys navigate the menu.  <Enter> selects submenus --->.  Highlighted letters
 are hotkeys.  Pressing <Y> includes, <N> excludes, <M> modularizes features.  Press
 <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] built-in  [ ] excluded
 <M> module  < > module capable

                  Processor type (ETRAX-100LX-v2)  --->
              (32) DRAM size (dec, in MB)
              (2) Buswidth of NOR flash in bytes
              (1) Buswidth of NAND flash in bytes
              (4) FLASH1 size (dec, in MB, 0 = Unknown)
                  Product LED port (None)  --->
                  Product debug-port (Serial-0)  --->
                  Product rescue-port (Serial-0)  --->
              (0x95f8) R_WAITSTATES
              (0x004) R_BUS_CONFIG
              [*] SDRAM support
              (0x09603737) R_SDRAM_CONFIG
              (0x80008002) R_SDRAM_TIMING
              (0x1d) R_PORT_PA_DIR
              (0xf0) R_PORT_PA_DATA
              (0x00) R_PORT_PB_CONFIG
              (0xce) R_PORT_PB_DIR
              (0x03) R_PORT_PB_DATA
              [ ] Software Shutdown Support

                   <Select>    < Exit >    < Help >
```

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

The principal calls are developed in a quite standard way. At the beginning there is a specific structure to menage the device named struct `gpio_private` containing many fields among which:

`next` pointer to the following item: this structure is in reality a list

`minor` device minor number port pointer char to the register of port (A, B, ecc)

`dir` pointer char to the register by which it is possible to set the "direction" of the port, or, to be more precise, if it is input or output shadow pointer char containing the value of port, that is the state of bits

`dir_shadow` pointer char to the value contained by the register pointed by dir field; remember that registers `R_PORT_P`x`_DIR` are write-only, so if we want to keep track of their values, we must use an auxiliary variable

`higalarm, lowalarm` are alarms that can be associated to the specific port; they are related only to the configured port as input and normally are managed in interrupt

This structure is loaded by *open* function that make it available to the other function. In order to do this the *open* obtains, from the file descriptor, the minor number through macro `MINOR(inode->i_rdev)`. If minor number is valid the execution proceed allocating in memory the variable `priv`, that is a pointer to the structure `gpio_private` and it set immediately the minor number inside of this structure. If port is A or B then the structure is compiled using some static arrays defined with the same name of the components of struct layout (`port`, `dir`, `shadow`, ecc but plurals).

Function *open* is called always when a device file is opened, and, as you know, every times that a port is opened, a new "device structure" is allocated, but this memory must be released just before the file descriptor of the device is released (in other words when a *close* function is called). To do this, function `gpio_release` is used and every operation executed inside of this function must be executed "blocking" the interrupt events; in other words the operations executed are written between two macro: `spin_lock_irq(&gpio_lock)` and `spin_unlock_irq(&gpio_lock)` (see chapter 5 of [4] for more informations). Memory user by each item of the list will be free.

At the and remaining only functions *write* and *ioctl*. Those function permit to access and communicate with the device. The first is implemented by `gpio_write` and execute the control of minor number and verify if buffer passed from user-space is valid for its whole length (`count`, also this passed from user-space) by macro `acces_ok`; the execution proceed blocking the interrupt event alike `gpio_release` and setting the value of the port in order to specific masks.

In order to `gpio_ioctl`, the behavior is similar, but it depends on the required function. In this case the access permit better interactions; indeed it is possible not only read and write a specific register of a port, but also read a single bit, set/reset alarms, etc.

This module has the only initialization function (`gpio_init`) that registers the module as a character device (`register_chrdev( GPIO_MAJOR, gpio_name, &gpio_fops)`). The code proceed configuring LED (and so the port) by a macro link `LED_NETWORK_SET`, `LED_DISK_READ`, etc according to the kernel configuration. Finaly the function tries to register two interrupt routine by `request_irq` as following:

```
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");
}
```

First *if* sets a function (`gpio_poll_timer_interrupt`) that should execute a cyclic control (due to the timer) of alarms eventually generated calling `etrax_gpio_-wake_up_check()`. The second *if* sets a function that should execute the same operation of the first one, but the trigger event is on the port A. "Should" because in reality this never happen! Those function aren't registered (or better, they are not assigned to the specific interrupt) because of some condition. One of these is the flag `SA_SHIRQ` which indicates the possibility to sharing the interrupt of this peripheral, but it isn't true, in particular for timer device; even if theoretically we could think that it is possible to share specific operation about temporization, probably it could generate unacceptable overhead also if the function is very fast (or, at least, atomic).

Curiously the module don't have an "unload" function. Probably it is due to the fact that the system doesn't schedule to load dynamically this module. But if the module would be separated by the monolithic kernel, it would be better to insert a function of this kind, at least to unregister the character device and to release the interrupt functions (if they are being registered).

2.3 Access from user space

Access to the I/O port through user-space is done besically in two ways: the first uses the typical device file abstraction method in UNIX, that is the access opening a specific device file, while the second permits the access through specific system calls.

2.3.1 IOCTL

The first method quoted before permits, as it said, to access through a device file contained in `/dev` directory. Normally on a device file it is possible execute the operations like open, read, write, etc. But in this case not all the typical functions are being (reasonably) implemented; it is simple to think that the *append* function is unnecessary to control a device like a digital port, and so on for many other system call, for instance *fseek*. Up ahead this fact will be more clear, when we will analyze the driver in details.

Then, abstract a digital port by a character device file is possible, all the more ports A and B are accessible by indexing 8 bit for per time. Nevertheless the best method to take the control of the port is using the system call *ioctl* which permit to execute more operations by which:

IO_READBITS read single bits of the port (in and out). This function is deprecated and it is suggested to substitute it with IO_READ_INBITS and IO_READ_OUTBITS.

IO_SETBITS sets state of one or more bit, then drive the corresponding pin of processor at high level.

IO_CLRBITS resets state of one or more bit, then drive the corresponding pin of processor at low level.

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.

In the follow it is showed an example taken by ACMESYSTEM site and readapted. This code gets the red LED blinking. As we saw before, the red LED is connected to the fourth pin of port A (PA3); so we have:

```
#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; //mask indicating the 4° bit, then 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;
}
```

This program set on and off the LED 20 times.

2.3.2 Syscall

The second method allowed to control the specified port through direct special system calls (syscall as you know). These calls are defined in file `linux/gpio_syscall.h` and with them we can have access to the kernel directly without using a device file; so we couldn't use a descriptor file. In other word we don't have to open a device file to use the specific system calls releted to the device. When a routine of a specific device is invoked (like *open* or *ioctl*) by a process in user-space, the execution passes in kernel-space by a software interrupt request[6] that normally need of many cycles typically to save the process context. Normally this kind of events get worst the performances of program, in particular if they are repeated many times. The syscall implemented in `gpio_syscalls.h` avoid this problem and so they are more efficients.

An example alike the last one:

```
#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 is RED LED
       printf("%d\n", (gpiogetbits(PORTA, PA3))?(1):(0));
       sleep(1);
       gpioclearbits(PORTA, PA3);
       printf("%d\n", (gpiogetbits(PORTA, PA3))?(1):(0));
```

[6] By interrupt 80 in the x86 systems with the instruction `int 0x80`

```
        sleep(1);
     }
   return(0);
}
```

2.4 Access form kernel space

At kernel level the access to GPIO port is relatively simple and it is similar to that happens in a typical firmware writing for a memory mapped micro-controller. Also in this case we must write proper values in a specific address of memory (register).

Considering again port A, we can read from datasheet [1] the configuration register for this device is located to the index `0xB0000030`. Again from [1] we get the name of register is `R_PORT_PA_SET`, that is the same we can find in configuration kernel. This is obviously a 32-bit register (word of this ETRAX 100LX processor), but bits used from the port PA are only the first 16: the first 8 bits identify the state of pins, while the 8 most significant (of the 16 less significant of the whole word) indicate to the processor if the corresponding bit is an input or an output. These two registers are named `R_PORT_PA_DATA`, `R_PORT_PA_DIR` and their respective address are `0xB0000030` and `0xB0000031`. The register `R_PORT_PA_DIR` is moreover write-only.

Following examples verify what we have just said. If who read don't have experience about kernel programing, it is suggested to read [4]. In order to module compilation, see appendix A.

2.4.1 Retrieve Informations

The proposal module read the address and value of register about PA port:

```
/** 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");

/** Module Loading Function **/
void infoPA_cleanup_module(void) {
//Nothing to do
}
/** Initializzation Function **/
int infoPA_init_module(void) {
//cast to int to avoid warning in compilation time
   printk(KERN_ALERT "\nIndex port 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 "\nValue port 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);
```

This module is simple and is made by the only two function of initialization and cleanup; the firs write in the ring buffer of kernel the indexes and values about port A. When module is unloaded, nothing will be executed because there is no allocated memory, definitions in `/proc`, syscall substitutions, etc. To verify the code we can connect to the board, loading the module and watching the last row wrote in `/var/log/message` file:

```
telnet 192.168.0.90         #typical way to connect to the FOX
insmod /mnt/flash/infoPA.ko #loading module stored in in /mnt/flash
tail /var/log/message       #reading kernel report
rmmod infoPA                #eventually remove module from kernel
```

Note: to specify the type of message and to indicate to the kernel that the message must be write in `/var/log/message`, we add before the string the macro KERN_ALERT. There are many other kind of macros related to the type of message; one of this other macro is KERN_INFO and also this macro put the message in the same file (`/var/log/message`). This is different from the behavior of the standard kernel in which (typically) the messages marked by KERN_INFO are wrote in `/var/log/message`, while KERN_ALERT write the message in `/var/log/syslog`[7].

2.4.2 Change status of LEDs

The next module is a little more complex. Now we wont to drive the kernel to change the state of red and yellow LED. In order to do this, one possible solution is using `/proc` filesystem. We will create a file in the subdirectory `calzo/` named `pa`. Writing this virtual file we change the state of output pins. The various functions will be explained in the follow.

"Completeness is enemy of clearness": so the following modules could be uncompleted or could have some bug, but they should be more simple to understand.

Declarations & Macro

```
/** CALZOLED.C **/
#include <linux/module.h>
```

[7] This is true in Slackware; in other distro the name of some files can change

```
#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");

/** Module Paramethers **/
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 - to avoid
                      //"Concurrency and Race conditions" [4] ch5
//Structure for /proc filesystem
struct proc_dir_entry *proc_cldir = NULL; //main directory
struct proc_dir_entry *proc_clpa  = NULL; //file for port PA
/** Definition for /proc filesystem **/
#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);
```

Two macro `module_init` and `module_exit` are implemented at the end of file.

Module initialization

```
int cl_init_module(void) {
   //Semaphore initialization
   init_MUTEX(&sem);
   //Creation of virtual directory /calzo in /proc
   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, //file name
                      0, //protection mask: default=0
                      proc_cldir, //parent dir
                      NULL, //file read function
                      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 loaded.\n"
             " PA data:      %x\n PA direction: %x\n",
             *R_PORT_PA_DATA, *R_PORT_PA_DIR);
      return 0;
}
```

This function initializes semaphores used inside the write function. Successively `calzo/` directory is created in `/proc` with proc_mkdir routine. If there aren't any problems it proceeds to the creation of virtual file pa (by `create_proc_read_entry`). Before exiting, values of PA port are wrote in kernel ring buffer.

Module Clean-up

```
void cl_cleanup_module(void) {
   // ERASING proc entity
   if(proc_clpa)
     remove_proc_entry(PROC_CL_PA, proc_cldir); //file and parent-dir
   if(proc_cldir)
     remove_proc_entry(PROC_CL_DIR, NULL);

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

Removing virtual file in `/proc` calling `remove_proc_entry`.

Write function for virtual file `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; //if new value is negative it's wrong

   if(!count)      //if counter is 0...
     return count; //...return 0 and don't do anything
   if(!buffer)     //if buffer doesn't exist...
     return 0;     //...0, but it would be better return an error
   if(down_interruptible(&sem)) //avoid concurrency conditions
     return -ERESTARTSYS;

//If we are here, we proceed with a copy of buffer
// passed from user-space for a maximum of length of
// destination buffer.
   copy_from_user((void*)&buf, buffer, 1);

//Convert HEXADECIMAL char in a real number
   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;

   //If there aren't errors, set data in port A
      if(new_value<0)
        printk(KERN_ALERT "Warning: %c is not HEX format.\n", buf);
      else
        *R_PORT_PA_DATA = new_value;

      up(&sem);     //release semaphore
      return count; //returning count, the whole process is done in a while
   }
```

Function *write* for file system *proc* has a parameter headline a little different by the traditional *write* call, but the concept is the same. After the typical initial checks, the kernel checks the first character of a buffer passed from user-space and transforms it a number if this character represents a hexadecimal number. Analyzing only one letter, the possible values that are writing in the register of port A will be between 0 and 15. In other words we can change only the first four bits. It is enough because FOX LED are connected to the first four pins.

Using the module

Once we have compiled the module and loaded on FOX (for example in `/mnt/flash`) it is necessary load it with the well-known command insmod. Therefore we find in file system `/proc` the directory `calzo/` and file `pa`. Now with the command:

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

the first four pin of port PA are set to the logical state high if they are output. LEDs which are controlled in negate logic shut off them. Vice versa writing `0` instead of `F`, LEDs turn on.

2.5 Interrupt

In this section we try to regiser and active an interrupt function aganinst an event on port A. In the section 2.2 was introducted the syntax by which is possible to join a function to the specific event. The code is:

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

but, as we have already mentioned, this function will never be register if there is `SA_SHIRQ` flag; if it is removed, the function `gpio_pa_interrupt` will be registered with success and we can see that typing `cat /proc/interrupt` (figure 3). At this point, for debugging, we can write for example `printk(KERN_ALERT "interrupt registered!\n")` at the beginning of `gpio_pa_interrupt` to see if the interrupt is invoked or not. If now we press the button on FOX Board we maybe think an interrupt request is generated and this event write "interrupt registered!" string in `/var/log/message`, but it isn't. We must do another thing that is not done by the kernel, to be more precise it needs to operate above `R_IRQ_MASK1_SET` register which indicates which pins of port A can generate interrupts. To do this we can modify the initialization function of *infoPA* module described before (or, if you prefer, you'll write another module) adding the code line `*R_IRQ_MASK1_SET=0xF` by which we set the interrupt event will be activated only from the firsts four pins of port A if they are configured as input; so with this configuration the only pin which could generate an interrupt is linked to the button.

But now there is a problem: suppose to connect and load the module which allows to enable the interrupt. At this point, without doing anything else, we notice a strange behavior of board that suddenly must satisfy a huge queue of interrupt requests! Pushing the button we see moreover the processor menage the queue and it is able to free some of requests, but, after some push, the kernel message comes over:

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

so the interrupt become disabled. Now (or better before) we can reboot the FOX board.

This behavior is all in all correct because from [2] we read interrupts on ETRAX 100LX processor related to the port PA rise "at level" high, so interrupts rise if the (voltage) level on a specific pin remain high. The button

Figure 3: `cat /proc/interrupt` output

```
        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)
```

The maximum number of interrupts allowed by ETRAX 100LX processor is 32; the limit is set in the software by "`#define NR_IRQS 32`" macro in `$AXIS_KERNEL_DIR/include/asm-cris/arch-v10/irq.h`.

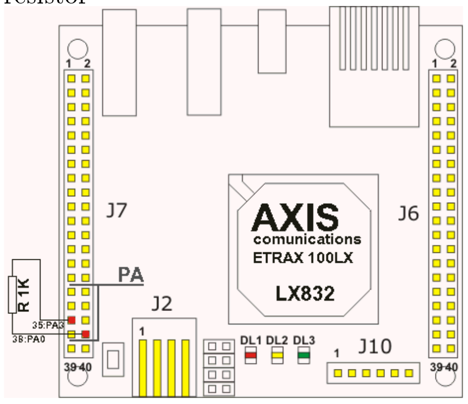
is connected in negate logic directly to the processor so, obviously, in the rest state (default state) detect the high level on this pin and generate an endless interrupt requests.

For these reasons, we will write a new module named `irqPA.c` to test interrupt.

2.5.1 irqPA

The objective of this module is detect an interrupt on one pin of port PA avoiding problems mentioned before. There are many way to do this, but we have decided to set as input another pin not connected to other device and we command it by an other pin configured as output in user-space by *setbits* command. If we examine the structure of the FOX about GPIO (see [5]), we see probably the better choice is to convert PA0 as input, while PA3 (red led) will be the controller (the last one choice is arbitrary). To connect these pins we can chose a resistor of about $1K\Omega$ as you can see in figure 4.

Figure 4: Pin PA0 (J7.38) and PA3 (J7.35) connected on FOX board with $1K\Omega$ resistor



PORT A: default connection
A0 J7.38 (I/O) O General purpose I/O
A1 J7.37 (I/O) I On board switch SW1
A2 J7.36 (I/O) O On board led DL2
A3 J7.35 (I/O) O On board led DL1
A4 J7.34 (I/O) O DTR line on /dev/ttyS2
A5 J7.33 (I/O) I RI line on /dev/ttyS2
A6 J7.32 (I/O) I DSR line on /dev/ttyS2
A7 J7.31 (I/O) I CD line on /dev/ttyS2

From the point of view of software configuration, we can reconfigure the kernel in order to set the first two pins of port A as output. To do that we run the kernel configuration command and we modify the item `R_PORT_PA_DIR` (figure 2) at value `0x1C`. Obviously the kernel must be recompiled and writed in flash. To verify if the modification is correct, typing the command readbits which returns as first characters (about port A) "`111XXX10`".

The module will be presented must execute this operations:

loading	registering with success the interrupt function when will be loaded
interrupt	while interrupt is running the routine code, it must drive down the logical state of pin PA3; this is mandatory to avoid, as already mentioned, that the system will be overcrowded by interrupt requests. Remember this is true only for this kind of example!

unloading while module run the clean-up routine it is necessary to deallocate the interrupt function

```
#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 for button

static DEFINE_SPINLOCK(gpio_lock_irq);

/***** INTERRUPT FUNCTION ********
* This function disalbe temporary the possibility to
* receive other interrupt until it isn't completed
* *******************************/
static irqreturn_t
  irqPA_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
   *R_IRQ_MASK1_CLR = 0;
   *R_PORT_PA_DATA  = 0; //necessary to avoid an avalance of interrupt
   printk(KERN_ALERT "calzo - irq served\n");
   return IRQ_HANDLED;
}
/****************************************
* INITIALIZATION & CLEANUP functions
****************************************/
/// Module initialization function
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;
     }
   //set which pin can generate the interrupt
   *R_IRQ_MASK1_SET = IRQ_PA_MASK;
   return 0;
}

///Module unload function
void irqPA_cleanup_module(void) {
   spin_lock_irq(&gpio_lock_irq);
   *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);
```

Note: in function `irqPA_cleanup_module` is used a function `spin_lock_irq(&gpio-_lock_irq)` which inhibit the interrupt temporary on the current processor. Maybe this isn't too much correct or however it can be unnecessary because we act with `R_IRQ_MASK1_SET` resetting itself, so disabling the interrupt of PA port.

Remember if you want to use this module is better to comment the registration of interrupt function in file `$AXIS_KERNEL_DIR/arch/cris/arch-v10/drivers/gpio.c`. In this manner you are sure that there aren't other registered function.

How to use this module

Once you are connected to the FOX and loaded the module with *insmod* command, if is not showed any error, you can control the module was correctly loaded typing `cat /proc/interrupt` that must return a message similar or equal that is showed in figure 3.

If there are no problems it is necessary to set high the PA3 pin typing "`setbits -p a -b 3 -s 1`". Doing this, the PA0 pin rise to the high level ant the interrupt is detected and served. Now you could think to see red LED turn off itself, but it isn't because at most a few tens of microsecond the interrupt routine will be served; this code proceeds resetting `*R_PORT_PA_DATA`, operation mandatory. Obviously it could be better to reset only the pin that generates the interrupt (PA0), but for precision and simplicity it is preferred to proceed thus.

Appendix

A How to configure&compile kernel module

What is described in the following sections is relative to the kernel 2.6.15 of FOX board, but it is applicable to any other kernel.

A.1 Configuring kernel modules

It can be necessary or however useful to insert in the configuration interface of the kernel, the possibility to choose the module just wrote and/or can configure them. In order to do this it needs a file named `Kconfig` in the directory containing new files that it would compile. It is evident if files are in a directory already present in the kernel tree, it will be sufficient to edit the `Kconfig` file that it is in. In this example it will consider always the `$AXIS_KERNEL_DIR/drivers/calzo` directory.

First we'll examine `Kconfig` within the directory `arch/cris/` of kernel tree which define sub-menus of the root of kernel configuration, that is the first screen you can view typing `make menuconfig`. Append to the file:

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

add the voice *Calzo Device* to the configuration interface which is a sub-menu. Now it is necessary to write the files `drivers/calzo/Kconfig` (also void) to avoid the fault of creation of the configuration interface. Adding to the file the following lines:

```
config CALZO_MODULES_ENABLE
       bool "Active Calzo Modules"
       help
         Some modules writing by Calzo for Linux DAY '07
         Linux Day 2007 come esempio

config CALZO_INFOPA
       tristate "infoPA"
       depends on CALZO_MODULES_ENABLE
       default m
       help
         infoPA return indexes and value of port A on ETRAX LX100
         Compiled as a module, the name is infoPA.ko

config CALZO_LED_PA
       tristate "Calzo LED"
       depends on CALZO_MODULES_ENABLE
       default m
       help
         Set the status LED of port A writing an hexadecimal char
         in /proc/calzo/cl. For instance echo F > /proc/calzo/cl
```

```
          shut off all LED.
          Compiled as a module, the name is calzoled.ko
```

Running the kernel configuration, activing these new options and saving them, will be created a new `.config` file containing for example:

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

As you know the configuration system will add these informations in a .h file as a C definition and also before these macros will be used to instruct the compiler about new files it has to compile.

A.2 Compiling kernel module

To compile a kernel module is better make a new directory in the root kernel source. In this case, guessing the sources of the whole system are in `/usr/local/fox`, move in the subdirectory `os/linux-2.6` that is the kernel source directory. Once here we can create for example a directory `drivers/calzo/` typing `mkdir drivers/calzo/`; so we can edit the `Makefile` within `derivers/` directory adding (for example to the bottom) "`obj-y += calzo/`".

So the kernel compilation system will know it'll have to find a `Makefile` inside the directory `derivers/calzo/` and so it will be necessary to create this file even if empty, otherwise the compilation process fails. In this makefile it needs to indicate which (new) modules has to be compiled. Wander to have two modules named `calzoled.c` and `infoPA.c` and adding:

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

These instructions allow to compile `calzoled.c` and linked it directly in the kernel, while `infoPA.c` will be compiled as a module and so in the its own directory will appear `infoPA.ko`; obviously extension `.ko` is due to the version 2.6 of the kernel.

In the section A.1 it saw how to configure a module. To make the new configuration usefull by the compiler, it must write:

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

where these macros have value `y` o `m` if you wont to add the module directly in the kernel or get it separated an load it later.

A non monolithic compilation of module is done by typing `make` in the directory of the kernel. Take in mind that to do a cross-compilation of any application of FOX, it is mandatory execute init_env in the root directory of FOX sources as mentioned in [5]:

```
cd /directory/of/fox/environment/devboard-R2_01
. init_env
```

The last command creates new environment variables to do the correct compilation with *cris* compiler. The variable is alike to this:

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

To compile is sufficient run `make` or `make modules` if the modifications are only of modules. if modules are compiled and added int the kernel code, it must reflash the FOX, while if modules are separated from the kernel code (so exist one or more `.ko` files), we can copy them with the `scp` command; normally:

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

Keep in mind that `/mnt/flash` is a directory always writable in flash.

Reference

[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 how to cross-compile a program

Info&Credits

Many thanks to **MCM Energy Lab** and the section of Azionamenti of the department of Electrical Engeneering of Politecnico in Milan (Italy) by giving support and hardware to do the test.

This document was written for the 7° days of Linux and Free Software and presented by **LUGMan** (`www.lugman.org`) in Sangiorgio (Mantova); it is released under GPL v2 license. Everyone would have the sources of this document can ask it to info@lugman.org or browsing the web site of the association. This document is writen with LyX 1.4.3 under GNU/Linux Slackware 10.2. All the software is writen and tested with GNU/Linux Slackware 10.2.

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

release 1 October 2007 for Linux DAY 2007 in Mantova (Italy) - written by Calzoni Pietro aka *Calzo*, member of LUGMan - Linux Users Group Mantova

release 2 March 2008 - translation by *Calzo*. Thank you very much for the interest showed me about this document by many people

release 3 August 2008 - some little correction. Special thanks to Geert Vancompernolle for feedbacks