Workshop on RealTime Systems

Abdus Salam ICTP, Trieste.

October 28–November 21, 2002

Lecture Notes
Collected Adventures in Device Driver Writing

This chapter contains lecture notes on a Linux device driver

Contents

Chapter 1

Collected Adventures in Device Driver Writing

by Ulrich Raich

1.1 Abstract

Writing a device driver is one of the most tricky things you can do in programming. The problem is that deep knowledge of the hardware you want to control is needed as much as deep knowledge about the operating system you want to write the driver for. While a software bug in a usual user level program is acquitted with a core dump, making a similar fault in system mode when implementing a driver will normally result in a system crash.

These lecture notes give a step by step introduction on how one should try to tackle the problems encountered in device driver writing. It traces the difficulties the author himself had when trying to provide the *Colombo board* driver. The full driver source code is available for study.

1.2 Introduction

It may be best to tell you right from the beginning:

Device Driver Writing is a tricky business!

This in fact was the first lesson I learned myself when preparing this series of lectures. I was very proud when I was attributed this course by ICTP because device driver writing has the reputation of being rather difficult. So I was thinking of a course explaining

- all the complicated data structures needed in order to hook up the device driver with the kernel
- the context switch from user to supervisor mode with all its details
- lots of computer science theory of why device drivers are important
- and of course all the details of interrupt and DMA driven device drivers
- connection of file system with block device drivers etc.

I think you see what I mean. *The Theory of Device Driver Writing* might have been the right title. Then I had the splendid idea that it might be good to actually write a driver myself before trying to explain to others how to do it. That was the moment when everything began to go wrong! I started to write the code some 3 months before the course and 2 weeks before I had to give my lectures the driver still did not work! (and of course the transparencies were not prepared either!). The goals of my course became much more modest and I ended up with a course that tells you about all the mischieves I encountered when trying to implement my device driver. No theory! No block device drivers and file systems. Just the story of how I finally managed to get my device driver going. Nevertheless (or perhaps just because the course is now much simpler!) I hope that you will get some insight of

- what a device driver is
- how it works
- and how it is connected to the operating system.

- You get the full source code of the driver
- you can use it
- and you may modify it as ever you like (taking the risk of crashing the system)

1.3 Generalities

When accessing any type of hardware several problems arise:

- Firstly there are many people who understand often rather complex electronics that make up computer interfaces and there are many people who write splendid software. Finding somebody who understands the operating system writes nicely structured and very robust code (software) and who is capable to read circuit diagrams, understands timing signals and can read datasheets of electronic chips is already a different business. It is therefore reasonable to isolate the code that needs to know all about the intricacies of the hardware and which in addition must be extremely robust (a small bug can bring the whole system down!) into a separate module. Think of a disk driver for example. This module is written and thoroughly tested once and can then be used by everybody.
- In a multi tasking system several concurrent tasks may want to use the same resource, e.g. a line printer and may generate a big mess!
- As already mentionned above, hardware access should only be given to trusted users since an error may easily blow the whole system. This is particularly true for multi user systems.
- Access to fixed memory locations is needed e.g. registers or memory in an interface or even specialized I/O instructions. When a device is capable of generating interrupts or of performing DMA then the story becomes even more complex: For interrupts the program context is changed: A new stack

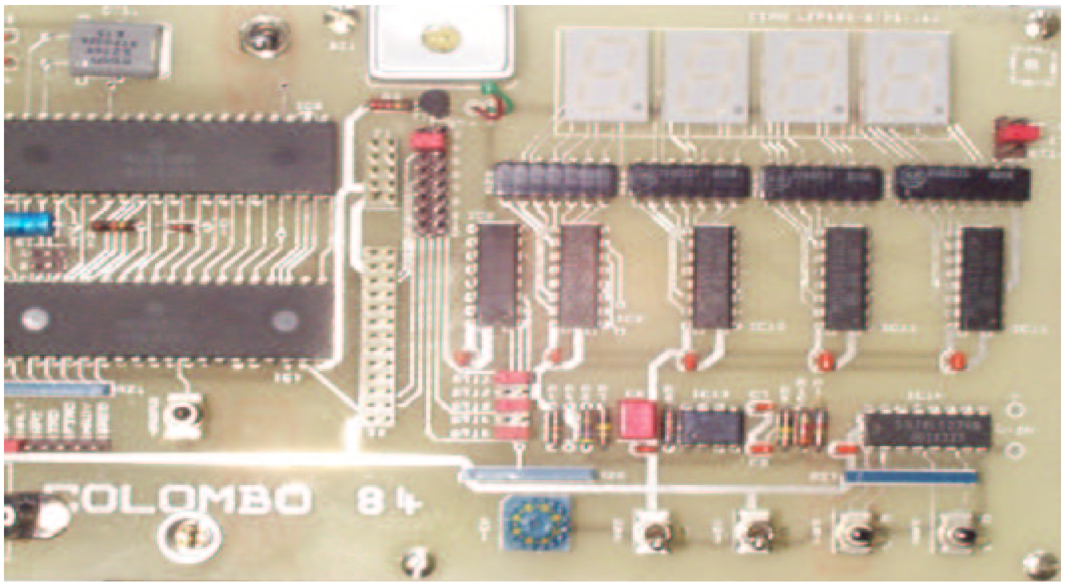

Figure 1.1: The Colombo Board

frame is in use and the CPU running mode is changed from user mode to supervisor mode. Therefore device drivers must very tightly cooperate with the operation system kernel.

1.4 Testing the Hardware

Going through the problems one by one I decided that understanding the hardware was first priority. So, what hardware should I use for my demo device driver? At ICTP the only easily available hardware was the Colombo board (1.1, which has been designed for the course 84 in Colombo (Sri Lanka). This board actually consists of 2 parts: a processor/memory/interface part (which in our case will be permanently disabled and which I will not describe any further) and a part simulating some sort of external process to be controlled. This I/O part consists of

- 4 hex seven segment displays (BCD displays in the original version)
- toggle buttons
- 2 pushbutton switches
- a rotary switch with 16 positions
- a voltage to frequency converter, allowing to simulate a simple ADC
- 3 fixed frequencies

The pushbuttons, the voltage to frequency converter and the fixed frequencies may be used to generate interrupts.

This board was designed to be hooked up to a PIA (Motorola M6821, Parallel Interface Adapter) in a M6809 development system with the processor part disabled. Development of programs could then be easily done on the development system using its assembler/linker/loader/debugger. Once everything worked fine a few addresses needed to be changed and the program was blown into EPROM and installed on the Colombo board. Enabling the processor part allowed to run the system in standalone mode.

As I said before, from now on we are only interested in the I/O part of the Colombo board. In order to use it for the device driver I still needed an interface for it on the PC (PIA equivalent).

When first playing with the idea of these driver lectures (some 10 years ago) I tried interfacing the Colombo board to my PC's parallel port and I managed to get at least the displays going. Then however I learned that a parallel I/O board had been developped at LIP in Portugal.

The I/O board, the General Parallel Interface (GPI) consisted of an Intel 8255 parallel input/output port, known under the name "Programmable Peripheral Interface (PPI)" and some interface logic connecting the 8255 to the ISA PC bus. It was for this board that the ICTP driver was originally developed and we run this driver, with various modifications due to kernel changes and with various improvements over the last ten years.

However, PC technology has changed and the ISA bus was replaced by the PCI bus. Up to a few years back there was always at least one ISA slot on the PC backplanes for backward compatibility but newer PCs have stopped having this feature. Since ICTP uses these new PCs which do not support the ISA bus any more, the I/O board had to be abandonned. We needed another solution!

The easiest and quickest solution was to come back to the parallel port. This port had been designed for interfacing to a printer but has evolved over time to be a multi-function interface that allows reading as well as writing.

The pinout of the parallel port is shown in fig. 1.2, page 5. Comparing this with the signals needed for the Colombo board it becomes clear very quickly that the data lines will need to be mapped to the display, while the rotary switch needs to get connected to the printer status lines somehow, or we must make use of extended features of the parallel port using it in bi-directional mode. Since most modern PCs use bi-directional parallel port (and we abandonned the GPI interface because it was too old-fashionned) I opted for this solution.

Accessing the pins, I mean sending signals to the data lines or reading back the status information is done through 3 registers shown in table 1.1.

Table 1.1: Addressing the PC parallel port

A0	**A1**	**Function**
0	0	Data register
0	1	Status register
1	0	Command register

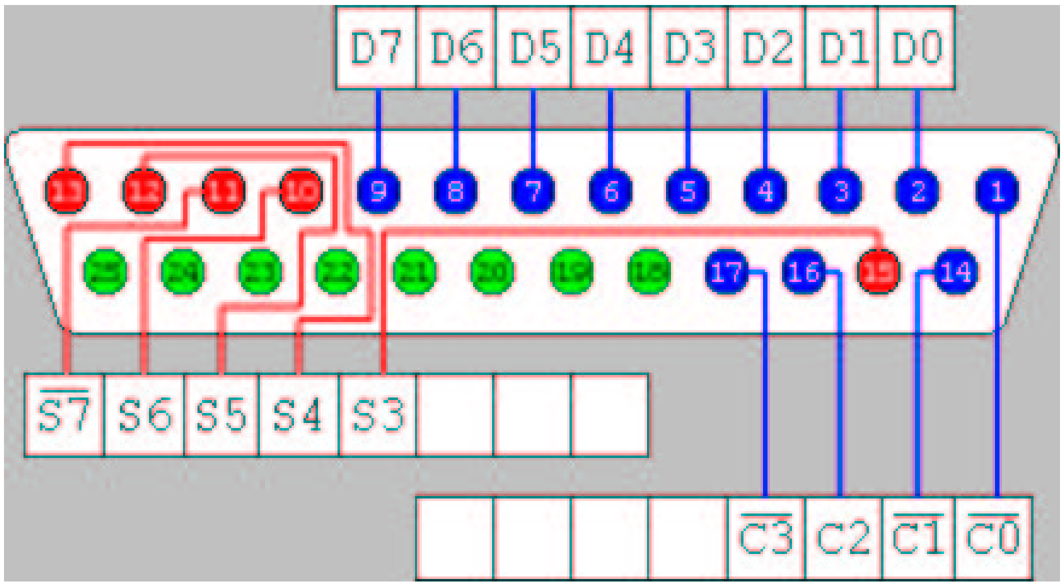

Figure 1.2: The Parallel Port Pinout

Table 1.2: The PC parallel port data register

Offset	**Name**	**Read/Write**	**Bit No.**	**Properties**
Base + 0	Data Port	Write	Bit 7	Data 7
			Bit 6	Data 6
			Bit 5	Data 5
			Bit 4	Data 4
			Bit 3	Data 3
			Bit 2	Data 2
			Bit 1	Data 1
			Bit 0	Data 0

The designation of the individual register bits can be found in tables 1.2 — 1.4.

Of course the pinout of the parallel port is incompatible with the Colombo board. The parallel port uses a 25 pin Cannon connector while the Colombo board connects through a Scotchflex connector. This problem must be solved with an adapter board. The adapter needs just 2 driver chips multiplexing the switch datalines and the display datalines from the Colombo board to the bi-directional parallel port datalines. The parallel port's strobe line is used for switching between writing the displays (strobe signal low) and reading the switches (strobe signal high).

Table 1.3: The PC parallel port status register

Offset	**Name**	**Read/Write**	**Bit No.**	**Properties**
Base +1	Status Port	Read Only	Bit 7	Busy
			Bit 6	nAck
			Bit 5	Paper Out
			Bit 4	Select In
			Bit 3	Error
			Bit 2	nIRQ
			Bit 1	Reserved
			Bit 0	Reserved

I finally had the setup shown in fig. 1.4.

The hardware was complete and testing could start. Most modern PCs implement the parallel port in an ASIC chip which is directly installed on the PC's mother board. If you get something wrong in your adapter board you can easily kill the motherboard. Since I do not have enough money left for a new PC, I decided to restart my old I486 PC (the one I did my original tests on, some 10 years back!) and I used its less functional port in order to do at least the preliminary tests. On this PC the parallel port was still implemented on a plugin card and the risk was therefore much lower. On the other hand only the output part could be tested since the board did not support bi-directional data transfers.

Table 1.4: The PC parallel port control register

Offset	Name	Read/Write	Bit No.	Properties
Base +2	Control Port	Read/Write	Bit 7	Unused
			Bit 6	Unused
			Bit 5	Enable Bi-Directional Port
			Bit 4	Enable IRQ via Ack Line
			Bit 3	Select Printer
			Bit 2	Initialize Printer
			Bit 1	Auto Linefeed
			Bit 0	Strobe

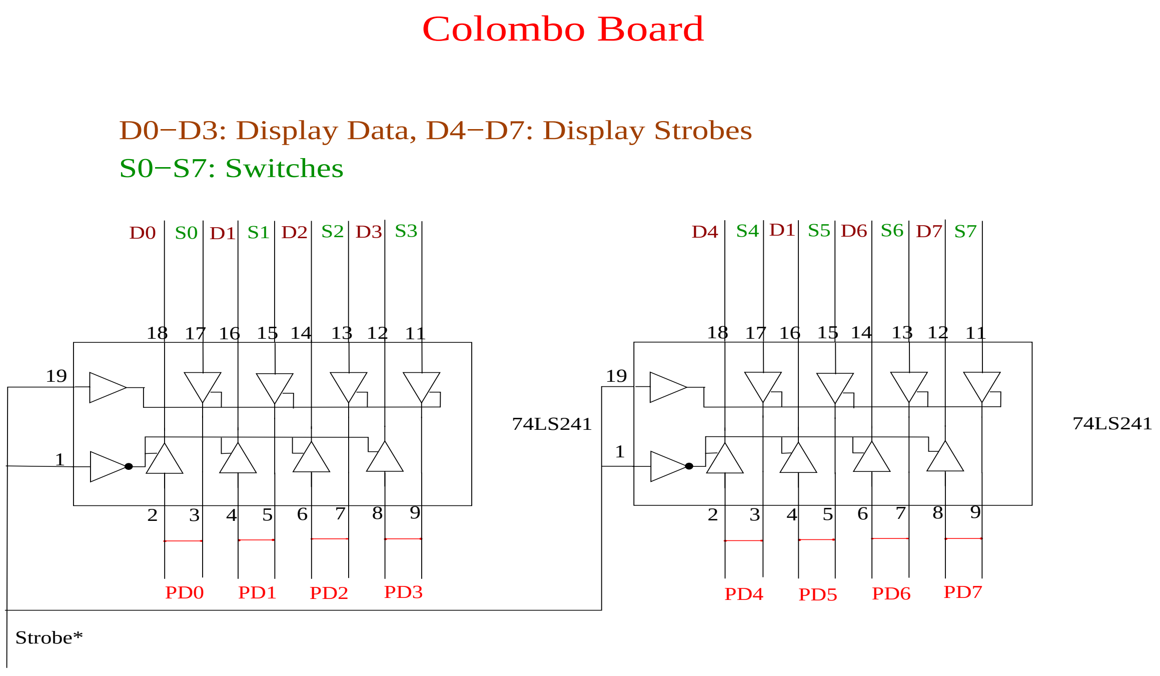

Figure 1.3: Adapting the Colombo board signals to parallel port signals

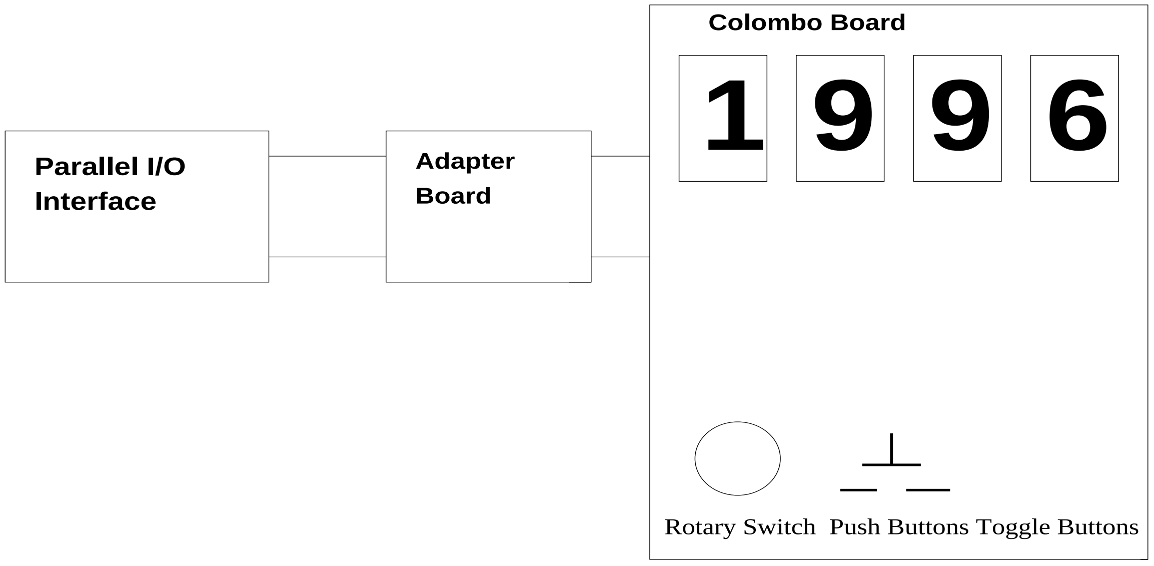

Figure 1.4: The Hardware Setup

Table 1.5: I/O Addresses on the PC Bus

I/O Addresses	Device
000-01F0	DMA Controller 1
020-03F	Interrupt Controller 1
040-05F	Timer
060-06F	Keyboard
070-07F	Realtime Clock
...	some left out
1F0-1F0	Fixed Disk
278-2FF	Parallel Printer Port 2
2F8-2FF	Serial Port 2
300-31F	Protytype Card
378-3FF	Parallel Printer Port 1
3F0-3F7	Floppy Disk Controller

I collected all information I was given on my PC and I found the above address table telling me that the parallel port can be found on the I/O address 0x378.

How can we talk to the parallel port? How do I know if I was able to access the board or not. Which calls are provided within Linux to access external hardware?

The main problem of checking the interface for the first time is to disentangle hardware and software problems. It is therefore important to get an indication that something works. The first thing to be tried is accessing the data register on the parallel port and reading the value back. This does not tell you anything about your external hardware or the connections you made, but it tells you if you are able to get hold of the interface registers. (On the parallel I/O port from LIP this was even more important since this board is not part of a standard PC). Of course checking out the hardware also means writing of very small and simple programs and this is where the problem starts! Hardware access is of course not allowed for any normal user (who would be able to entirely wreck our nice Linux operating system) but only the super user can do this. Now you may say: "...but I can access my files, the keyboard ..." which is entirely correct if you use the functionality of device drivers that have been installed by the superuser. More on this later on. In order to get any i/o access we (the superuser!) must first ask the operating system for permission which is done with the ioperm system call:

```
int ioperm(unsigned long from, unsigned long num, int turnon);
```

The first parameter contains the port's base address (0x378 in case of the first parallel port), num is the number of registers we want to access (3 in our case) and turnon is a boolean variable for switching permission on and off (true for switching on). The call returns a negative value if something goes wrong, e.g. you call without being superuser. Once access is permitted, we can write to and read from the parallel port registers using the macros:

```
outb(unsigned char value,unsigned long port_address); unsigned
char inb(unsigned long port_address);
```

How does the full program then look like? Here it is:

```
/*
 * hardware test for the PC parallel port in order to be used
 * as interface for the Colombo board
 * U. Raich 9-Oct-02
 * must be run as superuser!
 */

#include <sys/io.h>

int main(int argc, char **argv)
{
  int           retCode;

  unsigned char out_value,in_value;
  retCode = ioperm(0x378,3,1);

  if (retCode < 0)
    {
      perror("ictpuser: Could not get access to io port");
      exit(-1);
    }

  out_value = 0x55;
  outb(out_value,0x378);
  in_value = inb(0x378);
  printf("value read back: %x\n",in_value);
}
```

You can compile the program with

```
gcc -O2 -o first first.c
```

This program actually works nevertheless it is all but elegant. Firstly we should never compile programs without writing a Makefile. Makefiles can be extremely simple and they have the virtue of describing how programs are put together such that years after having developed a program you can easily re-build it by just typing “make”. Here is the Makefile for our test program:

```
CC=gcc
CFLAGS=-O2
all: first ictpin ictpout ictpstatus
clean:
rm -f *~ *.o core first ictpin ictpout ictpstatus
```

Secondly all numbers should be replaced by symbols, defined in an include file, which can later on be re-used by the final driver and by applications using the driver. The include file defines names for the different register offsets but also names for each control and status bit.

```
/********************************************************************/
/* include file for the ICTP device driver                          */
/* mainly defines hardware registers and bits of the parallel port  */
/* U. Raich 9.10.02                                                 */
/********************************************************************/

#ifndef TRUE
#define TRUE 1
#endif

#ifndef FALSE
#define FALSE 0
#endif

#define DEFAULT_MESSAGE_LOGLEVEL KERN_DEBUG
#define ICTP_MAJOR             0
#define ICTP_NO                3
#define ICTP_IRQ               7

#define BASE_ADDRESS 0x378
#define ICTP_IO_PORT BASE_ADDRESS
#define PP_DATA      BASE_ADDRESS
#define PP_STATUS    BASE_ADDRESS+1
#define PP_CMD       BASE_ADDRESS+2

/*
 * status register bits
 */

#define PP_IRQ        0x4
#define PP_ERROR      0x8
#define PP_SELECT_IN  0x10
#define PP_PAPER_OUT  0x20
#define PP_ACQ        0x40
```

```
#define PP_BUSY       0x80


/*
 * cmd register bits
 */

#define PP_WRITE      0x01
#define PP_AUTO_LF    0x02
#define PP_INIT_PRINT 0x04
#define PP_SEL_PRINT  0x08
#define PP_EN_IRQ     0x10
#define PP_BI_DIRECT  0x20
```

Using the include file the same program becomes much more readable:

```
/*
 * hardware test for the PC parallel port in order to be used
 * as interface for the Colombo board
 * U. Raich 9-Oct-02
 * must be run as superuser!
 */

#include <sys/io.h>
#include <ictp.h>

int main(int argc, char **argv)
{
  int           retCode;

  unsigned char out_value,in_value;
  retCode = ioperm(BASE_ADDRESS,3,TRUE);

  if (retCode < 0)
    {
      perror("ictpuser: Could not get access to io port");
      exit(-1);
    }
  out_value = 0x55;
  outb(out_value,PP_DATA);
  in_value = inb(PP_DATA);
  printf("value read back: %x\n",in_value);
}
```

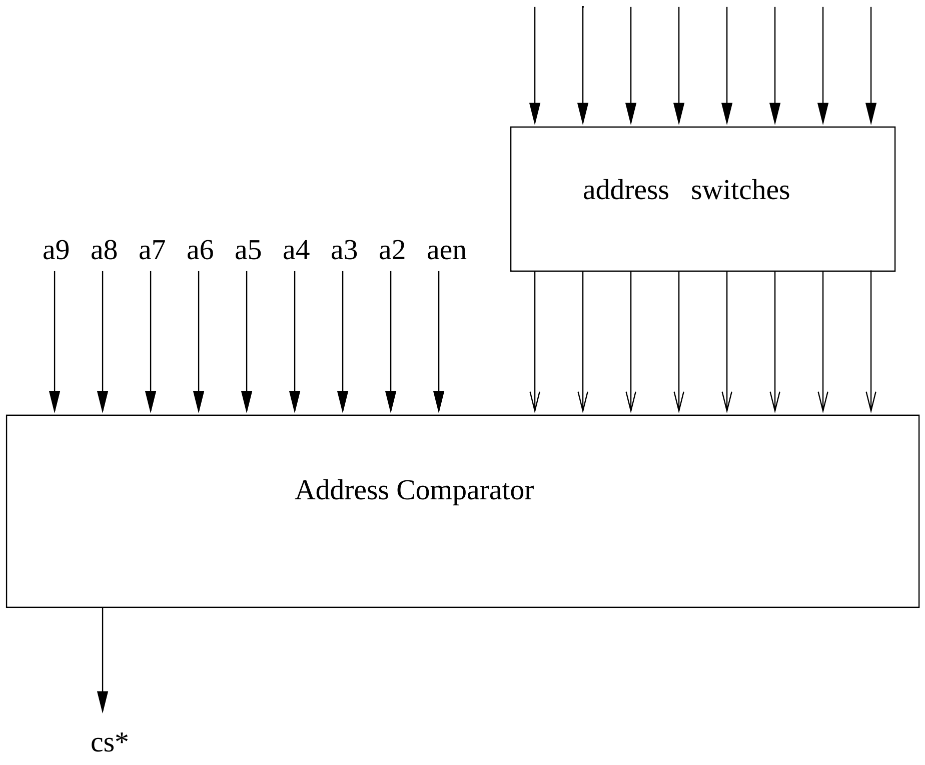

Figure 1.5: Address Selection

`inb, outb` etc. are macros, which are defined in /usr/include/linux/asm/io.h (please have a look at this file !) together with similar ones like inw, inl etc. Since these are "builtin macros" you **must** use the gcc option `-O2` in order to get them included into your code. Forgetting `-O2` results in unresolved references at link time.

Since the print statement resulted in some output giving the expected result I knew that I was at least able to access the port registers.

This one was easy! However when trying the same thing with LIP's GPI ISA board things were much more tricky; the board had an address decoder with DIP switches which allowed to select its base addess (see fig. 1.5

I had to make sure that I would not produce an address clash with other devices when setting up the DIP switches. I found in the I/O address table (table 1.5) that lpt2 (address 0x278) was not used and I set the switches to 0x13b (0x278 shifted left by 2 and 1 added for aen). I had to break out the metal protection for the I/O slot and I inserted the ISA card into the slot I had selected. I connected the cables and the decisive moment had come! I felt quite nervous! Another serious check and...

I switched the PC **on**!

Oufff, there was no smoke coming out of the PC and the thing seemed to be booting normally! However I quickly found that the floppy did not work any more! Did I finally kill the floppy interface or the floppy itself? I re-opened the PC (and decided to leave it open until everthing would work or I had to take the PC to the repair shop) and I took the GPI card out of the PC backplane. I tried booting again. Now the floppy worked correctly again. So, there must have been an address clash between the floppy interface and the GPI board. (I still don't know exactly how this came about!). Looking at the addresses again I selected 0x300 which was marked "prototype board'. (In Trieste this gave problems with the Ethernet card and the address had to be changed again!) The actual switch setting therefore was 0x181. I re-inserted the board and re-booted. Again the boot worked fine and now the floppy also worked normally.

However when trying our little program that accesses a register on the card, I never read back the data value of 0x55 I expected! This meant that I was unable to access the GPI board's registers. In the end I decided that there was probably an error in the circuit diagram and that the enable signal should be active low (as usually is the case). I therefore modified the switch settings to 0x180 and now, what a miracle, the little access program worked as expected.

Back to our parallel port interface. Unfortunatly we are still not at the end of our pain. All we are able to do up to now is to access some registers but we still do not see any effect of the external hardware. The next step should therefore give a clear indication that the adapter board is connected to the parallel port. The easiest thing to do is the connection of an output signal to a status line of the parallel port. We can then read back what had been written before.

```
/*
 * Write a bit to the output port and
 * read it back on the status lines
 *
 * U. Raich 9-Oct-02
 * must be run as superuser!
 */

#include <sys/io.h>
#include <ictp.h>

int main(int argc, char **argv)
{
  int           retCode;

  unsigned char out_value,in_value;
  retCode = ioperm(BASE_ADDRESS,3,TRUE);

  if (retCode < 0)
    { perror("ictpuser: Could not get access to io port");
      exit(-1); }
  out_value = 0xff;
  outb(out_value,PP_DATA);
  in_value = inb(PP_STATUS);
  printf("status read back: %x\n",in_value);
  /* wait a second */
  sleep(1);
  out_value = 0x0;
  outb(out_value,PP_DATA);
  in_value = inb(PP_STATUS);
  printf("status read back: %x\n",in_value);
}
```

Since this was working fine as well, I was sure that I could talk to the interface the way I expected. Now the final step; a program that has a visible effect on the Colombo board. I wanted to write to the displays and to read back the switches.

In order to write to a display on the Colombo board we must present a data nibble to all display registers with all strobe signals high, then present the same data with one strobe signal low (and the others high) and again present the data with the strobe signals all high. This will generate a low to high transition on the strobe line of one of the display registers thus clocking the data into the register. The data are then visible on the seven segment display.

The sequence of data bytes to be sent to the parallel port is therefore

```
0x3f 0x3e 0x3f
```

in order to write “3” to the rightmost display.

The following program writes a few numbers to the displays:

```
/*
 * Writes 1234 to the displays of the Colombo board
 *
 * U. Raich 9-Oct-02
 * must be run as superuser!
 */
#include <sys/io.h>
#include "ictp.h"

int main(int argc, char **argv)
{
  int           retCode;
  unsigned char value;

  retCode = ioperm(BASE_ADDRESS,3,TRUE);
  if (retCode < 0)
    { perror("ictpuser: Could not get access to io port");
```

```
      exit(-1); }

      value = PP_WRITE;         /* pull strobe signal low */
      outb(value,PP_CMD);

      value = 0x4f;
      outb(value,PP_DATA);
      value = 0x4e;
      outb(value,PP_DATA);
      value = 0x4f;
      outb(value,PP_DATA);

      value = 0x3f;
      outb(value,PP_DATA);
      value = 0x3d;
      outb(value,PP_DATA);
      value = 0x3f;
      outb(value,PP_DATA);

      value = 0x2f;
      outb(value,PP_DATA);
      value = 0x2b;
      outb(value,PP_DATA);
      value = 0x2f;
      outb(value,PP_DATA);

      value = 0x1f;
      outb(value,PP_DATA);
      value = 0x17;
      outb(value,PP_DATA);
      value = 0x1f;
      outb(value,PP_DATA);
}
```

And finally the program that reads the switches

```
/*
 * hardware test for the PC parallel port in order to be used
 * as interface for the Colombo board
 * reading the switches
 * U. Raich 9-Oct-02
 * must be run as superuser!
 */
#include <sys/io.h>
#include "ictp.h"
```

```
int main(int argc, char **argv)
{
  int           retCode;
  unsigned char value;

  retCode = ioperm(BASE_ADDRESS,3,TRUE);
  if (retCode < 0)
    {
      perror("ictpuser: Could not get access to io port");
      exit(-1);
    }

  value = PP_BI_DIRECT;            /* set to input mode */
  outb(value,PP_CMD);

  for (;;)
    {
      value = inb(PP_DATA);
      printf("value: %x\n",value);
    }
}
```

1.5 Accessing a device driver

You may think, since we now have access to our I/O card, we can read from and write data to it, well ..., that's it, we have finished! Unfortunately this is not the case. As said before: Any program making use of ioperm, inb, outb or mmap will only run in super user mode. We want to give access to our board to the ordinary user however. In addition there is no resource protection (the board may be written to by several tasks in any wild sequence) and treatment of interrupts or even DMA are excluded. Only the device driver will give you access to these possibilities.

What exactly is a device driver then? and how may an ordinary user access it? We want to slowly approach this question by first looking at the drivers software interface, or said differently: the way a programmer would use the driver.

You have already written programs that make use of files and you have seen the calls:

- open
- close
- read
- write
- lseek etc.

Accessing a device driver is exactly the same. You may think of a device as a **special file** (which is actually the technical term for it). The device is accessed through inodes defined in /dev using the same calls as normal file access.

The parallel port is somewhat special because there are many different devices that can be connected through this type of interface. Not only the Colombo board may be driven through the parallel port but also printers, scanners and even hard disks! Since the register layout is well defined it seems logical to write a *lowlevel driver* talking to the parallel port registers without taking into account any specific protocol features a scanner or printer (or the Colombo board) may need. This low level driver named ppdev, will then be used by higher level software implementing the specific protocol.

Linux kernel hackers already provide such a low level parallel port driver that can be called using the standard io calls described above. We want to use this driver in order to show how driver access is accomplished.

During this lecture series we want to learn how to write device drivers in general and not be restricted to the parallel port. For this reason the demo driver implements a full fletched driver which can be accessed in a way *very* similar to what we will show now (except that writing the data will be done with write calls instead of ioctl, see the routine out_digit).

In order to open the device of our Colombo board we would write:

```
fd = open("/dev/parport0",O_RDWR);
```

The open call returns an integer file descriptor which will be used as a parameter for all subsequent calls to the driver.

Once this is done we have to call a driver function claiming access to the parallel port we are connected to. Some other device may be using it already! This is done with an ioctl call:

```
  if (ioctl(fd,PPCLAIM)) {
    perror("PPCLAIM");
    close(fd);
    return -1;
  }
```

The symbolic names for the ioctl commands are found in the include file `ppdev.h` in /usr/include/linux. What exactly happens when we do this? The ”calls” open, read, write are so-called system-calls and differ from normal subroutine calls. System calls generate software interrupts and in doing so change the running mode from (normal) user mode to supervisor mode. After that a subroutine within the system kernel is called and executed in supervisor mode (compare to fig. 1.6). You now immediately see that our driver routines are actually executed on the same level as the kernel, they are integral part of the kernel. This also means that errors within the kernel routines are usually unrecoverable (there is no such things as ”segmentation fault, core dumped”) and will crash the entire system.

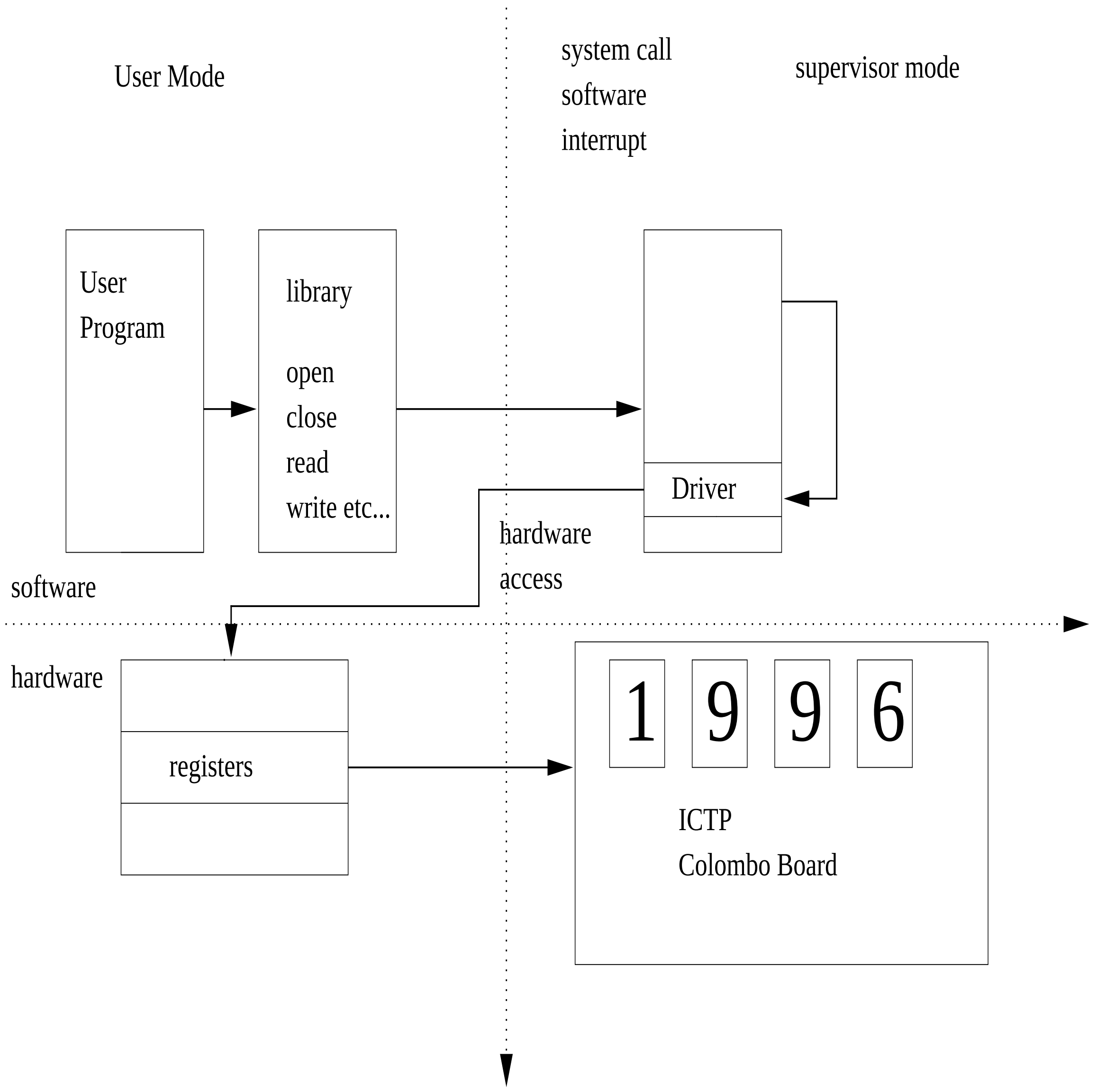

Figure 1.6: Accessing the Device Driver

The following listing gives an example of access to the ppdev driver sending some data to the displays.

```
/**********************************************************************/
/* A user level driver providing read and write call to the ictp    */
/* port. It uses the parport device ppdev                             */
/* U. Raich 14.10.02                                                  */
/**********************************************************************/

#include <fcntl.h>
#include <stdio.h>
#include <libio.h>
#include <unistd.h>
#include <asm/ioctl.h>
#include <linux/ppdev.h>

#define PP_WRITE 0x01
/*
  output data to a Colombo board display digit
*/

static void
out_digit(int fd, unsigned char digit, unsigned char number)

{
  unsigned char mask,c;

  printf("data: %x, digit num: %d\n",digit,number);
  c = PP_WRITE;

  ioctl(fd,PPWCONTROL,&c);
  mask = 1 << digit;
  mask = ~mask;
  c = (number << 4) | 0xf;
  ioctl(fd,PPWDATA,&c);

  c &= mask;
  ioctl(fd,PPWDATA,&c);

  c |= 0xf;
  ioctl(fd,PPWDATA,&c);

}
```

```
int ictp_write(unsigned short data)
{
  int i,fd;
  unsigned char pp_data;

  /*
    open the device driver
  */

  fd = open("/dev/parport0",O_RDWR);
  if (fd < 0)
    {
      perror("Cannot open parport");
      return -1;
    }
  else
    printf("ppdev successfully opened!\n");

  /*
    ask for permission to access the parallel port
  */

  if (ioctl(fd,PPCLAIM)) {
    perror("PPCLAIM");
    close(fd);
    return -1;
  }

  for (i=0;i<4;i++)
    {
      pp_data = data & 0xf;
      out_digit(fd,pp_data,i);
      data = data >>4;
    }
  /*
    release the port for other use
  */
  if (ioctl(fd,PPRELEASE)) {
    perror("PPRELEASE");
    close(fd);
    return -1;
  }
  close(fd);
}
```

```
int main (int argc, char **argv)
{
  short data = 0x0123;
  ictp_write(data);
}
```

1.6 Representation of the device driver

Having seen how to access the driver from an application we must now figure out how the kernel finds its way into the driver. Trying:

```
ls -l /dev/ictp*
```

will produce the following output:

```
crw-rw-rw-   1 root     root      254,   0 Jan  4 19:07 /dev/ictp0
crw-rw-rw-   1 root     root      254,   1 Jan  4 19:07 /dev/ictp1
crw-rw-rw-   1 root     root      254,   2 Jan  4 19:07 /dev/ictp2
```

where c tells us that the file is actually a character device driver, rw are the usual read and write permission bits and 254 is the major and 0,1,2 the minor device numbers. These numbers are unique in the system. By the way: the device special files are not created by a text editor but by the command:

```
mknod /dev/ictp c 254 0
mknod /dev/drivername, device type, major number, minor number
```

The major number defines the I/O device, the minor number usually indicates a channel number (a serial I/O device may have 4 UARTs representing 4 serial I/O channels, which are driven by a single software module).

As explained before, the driver is an integral part of the operating system and is usually linked into the kernel during system generation. However a software package has been developped for Linux, allowing us to install and de-install device drivers (or other ”modules” like file systems etc.) into a running kernel. This **modules package** provides the following basic programs:

- *insmod*: install a module into the kernel
- *lsmod*: list all installed modules
- *rmmod*: de-install a module from the kernel
- *ksyms*: list exported kernel sysmbols

and the newer versions of the package have in addition:

- *modprobe*: same as insmod but a standard path is searched for the modules while insmod needs the full path name.

- *kmod*: allows demand loading of modules. The module is simply installed in a standard directory. When an application program calls a device driver routine (ex. ”open”) and the device driver does not exist in the system, a request to load the corresponding driver is sent to kmod. kmod loads the driver via modprobe and the application can continue.

The system you are currently using has all its modules installed in /lib/modules/2.4.18-3.

1.7 Implementing the Device Driver, first steps

A device driver always consists of at least 2 files:

- The driver include file (ictp.h).
- The driver code itself (ictp.c).

The include file contains the definitions of hardware addresses, register names, and names for each and every bit used within the I/O chip registers. In addition it contains definitions for error codes, names for driver operating modes, ioctl request names and the like. It is used by the driver itself, but it is usually also included by any program using the driver.

The following listing shows the include file provided for the ictp driver:

```
/*********************************************************************/
/* include file for the ICTP device driver                         */
/* mainly defines hardware registers and bits of the parallel port  */
/* U. Raich 9.10.02                                                 */
/*********************************************************************/

#ifndef TRUE
#define TRUE 1
#endif

#ifndef FALSE
#define FALSE 0
#endif

#define DEFAULT_MESSAGE_LOGLEVEL KERN_DEBUG
#define ICTP_MAJOR             0
#define ICTP_NO                3
#define ICTP_IRQ               7

#define BASE_ADDRESS 0x378
#define ICTP_IO_PORT BASE_ADDRESS
#define PP_DATA      BASE_ADDRESS
```

```
#define PP_STATUS    BASE_ADDRESS+1
#define PP_CMD       BASE_ADDRESS+2

/*
 * status register bits
 */

#define PP_IRQ        0x4
#define PP_ERROR      0x8
#define PP_SELECT_IN  0x10
#define PP_PAPER_OUT  0x20
#define PP_ACQ        0x40
#define PP_BUSY       0x80


/*
 * cmd register bits
 */

#define PP_WRITE      0x01
#define PP_AUTO_LF    0x02
#define PP_INIT_PRINT 0x04
#define PP_SEL_PRINT  0x08
#define PP_EN_IRQ     0x10
#define PP_BI_DIRECT  0x20

#define ICTP_MODE_BLOCKING     0
#define ICTP_MODE_NON_BLOCKING 1

#define ICTP_MODE_RAW          0
#define ICTP_MODE_SINGLE_DIGIT 1
#define ICTP_MODE_FULL_NUMBER  2
#define ICTP_MODE_SWITCHES     3
#define ICTP_MAX_WRITE_MODE    ICTP_MODE_SWITCHES

#define ICTP_SIMULATION        0
#define ICTP_HW                1

#define ICTP_READ_SWITCHES     0
#define ICTP_READ_IRQ_COUNT    1

/*
  the ioctl codes:
*/
```

```
#define ICTP_BUZZER_BIT        PP_INIT_PRINT
#define ICTP_BUZZER_ON         1
#define ICTP_BUZZER_OFF        0

#define ICTP_SET_WRITE_MODE    IOC_IN  | 0x0001
#define ICTP_GET_WRITE_MODE    IOC_OUT | 0x0001
#define ICTP_SET_READ_MODE     IOC_IN  | 0x0002
#define ICTP_GET_READ_MODE     IOC_OUT | 0x0002
#define ICTP_SET_RUN_MODE      IOC_IN  | 0x0003
#define ICTP_GET_RUN_MODE      IOC_OUT | 0x0003
#define ICTP_SET_BUZZER        IOC_IN  | 0x0004
#define ICTP_GET_BUZZER        IOC_OUT | 0x0004
#define ICTP_SET_SWITCHES      IOC_IN  | 0x0005
#define ICTP_GET_DISPLAY       IOC_OUT | 0x0005

#define ICTP_ENABLE_INTERRUPT            0x0006
#define ICTP_DISABLE_INTERRUPT           0x0007
```

As you can see, all addresses are calculated relative to a single base address *io_port*. This variable can be set to the parallel port base address using an option (base=0x278) to insmod. Like this the driver code becomes independant from the hardware addresses and is configurable for any system.

Before a user program can access the driver it must be included into the system. As already mentioned earlier, this can be done either by linking it into the kernel at system creation or we register the driver with the operation system once the module containing the driver gets installed with insmod. Therefore we must provide 2 routines:

- *init_module*, which is called by insmod and which will check if the parallel I/O card can be accessed at the base address specified before asking the kernel to register the *ictp* device driver.
- *cleanup_module*, which is called by rmmod and which cleanly removes the module from the system.

The init_module function contains a call to ictp_detect which will autodetect if the adapter board is connected to the parallel port. This is done by connecting an output bit back to a status line such that a change of state on the output line can be seen when reading the port status. If the adapter has been found the driver goes into ICTP_HW, the normal running mode, while otherwise it will run in ICTP_SIMULATION mode. In normal running mode ”writing to the switches” has no sense and is inhibited. In simulation mode the switches can be set (this simply sets an internal variable) and when reading the switches the previously written value is returned. Since in simulation mode the display can also be read we can write a program that entirely simulates the Colombo board with a driver.

Since the driver is an integral part of the operating system and works in supervisor mode, it has no access to the normal C library functions. It cannot be debugged with a normal debugger either (the debugger has no access to supervisory memory!). However a few calls are available to the device driver writer, one of which is *printk*, which is the kernel equivalent to *printf*.

For debugging purposes I therefore put a few printk statements in strategic places in order to be able to follow the execution of my code. When registering the device driver with the system the address of the *fops* table is passed as a parameter. This table contains the entry-points of the driver routines needed for the execution of

- open
- close
- read
- write
- lseek
- ioctl

and possibly a few more system calls. Here is the *fops* table of our ictp driver:

```
static struct file_operations ictp_fops = {
  read:     ictp_read,
  write:    ictp_write,
  ioctl:    ictp_ioctl,
  open:     ictp_open,
  release:  ictp_release,
};
```

The code for init_module is also given below:

```
/*
 * And now the modules code and kernel interface.
 */

int
init_module( void) {
  int result;

#ifdef ICTP_DEBUG
  printk(KERN_DEBUG "ictp:  init_module called\n");
  printk(KERN_DEBUG "ictp:  io base address %x\n",io_port);
#endif

/*
  initialize the chip
*/
  ictp_reset();
  /*
    register the device driver with the system
  */
  result=register_chrdev(ictp_major, "ictp", &ictp_fops);

  if (result < 0)
    {
      printk(KERN_ERR "register_chrdev failed: goodbye world :-(\n");
      return -EIO;
    }
  if (ictp_major == 0)  /* dynamic major number allocation */
    ictp_major=result;

#ifdef ICTP_DEBUG
printk(KERN_DEBUG "ictp: driver registered!\n");
#endif
  if (ictp_detect())
    run_mode = ICTP_HW;
  else
    run_mode = ICTP_SIMULATION;
  return 0;
}

void
cleanup_module( void) {

#ifdef ICTP_DEBUG
  printk(KERN_DEBUG "ictp:  cleanup_module called\n");
#endif

  if (unregister_chrdev(ictp_major, "ictp") != 0) {
    printk("cleanup_module failed\n");
  }
#ifdef ICTP_DEBUG
  else
    printk(KERN_DEBUG "cleanup_module succeeded\n");
#endif
}
```

The first version of the driver registered a fops table with no entries at all. This version clearly cannot do anything, however it should be possible to test installation and de-installation into the system using *insmod* and *rmmod*. I expected to find the *printk* output on the xconsole and lsmod should allow to check proper installation. What did I find? Well, following Murphies laws, it was the worst possible result: lsmod told me:

```
  Module                        Size  Used by    Not tainted
  ictp                          6114   0
```

but I had no trace whatsoever of my *printk* statements. I was not really sure who was right: *lsmod* or the missing output from *printk*. After several hours of research and some poking around on the internet I found the email address of a *guru* who had written a device driver before. He told me I could check where to find the system console by trying:

```
  date > /dev/console
```

When I tried this, I found the output of the date on the xconsole as expected. Still I did not know, where the *printk* messages had gone. The other test I could find was to recompile the kernel and link my driver into the system. When booting the newly created system I saw the very first *printk* output on the system console but not the following ones.Of course I then checked the system log (/var/log/messages) in order to find out what had happened and there I found all the *printk* output I had expected on the console. The *printk* output went into the system log! After having added the lines

```
  # Send debug messages to the system console
  kern.debug                                       /dev/console
```

to /etc/syslog.conf I finally got the debugging messages where I wanted them, namely on the xconsole.

As you can see in the init_module code the default major number is “0” which allows the system to assign a major number of its choice to this device driver. It is the /proc filesystem that tells us more about currently supported devices:

```
cat /proc/devices
```

prints a list of the following kind:

```
  1 mem
  2 pty
  3 ttyp
  4 ttyS
  5 cua
  7 vcs
 10 misc
 14 sound
 29 fb
 36 netlink
108 ppp
128 ptm
129 ptm
162 raw
```

```
180 usb
226 drm
254 ictp                      <=========

Block devices:
  1 ramdisk
  2 fd
  3 ide0
  9 md
 11 sr
 12 unnamed
 14 unnamed
 22 ide1
 38 unnamed
```

from which we can tell that the major number 254 has been assigned to our device driver.

Since the creation of the device nodes in /dev depends on the major number assigned by the system we will need a shell script that will

- insmod the driver into the system
- determine the major number from /proc/devices
- create the corresponding device nodes
- set the correct group/permissions on the device node

Here it is:

```
#!/bin/sh
module="ictp"
device="ictp"
mode="664"

# invoke insmod with all arguments we were passed
# and use a pathname, as newer modutils don’t look in . by default
/sbin/insmod  ./$module.o $* || exit 1

# remove stale nodes
rm -f /dev/${device}[0-3]

major=‘awk "\\$2==\"$module\" {print \\$1}" /proc/devices‘

echo "major=" $major
mknod /dev/${device}0 c $major 0
mknod /dev/${device}1 c $major 1
mknod /dev/${device}2 c $major 2
```

```
# give appropriate group/permissions, and change the group.
# not all distributions have "staff"; some have "wheel" instead.

grep '^staff:' /etc/group > /dev/null || group="wheel"

chgrp $group /dev/${device}[0-3]
chmod $mode /dev/${device}[0-3]
```

Of course this script can only be run by the superuser. However, once the driver is installed in the system anybody (of group wheel) can access the driver.

Another interesting feature of init_module is the detection of the adapter board. On the adapter board an output line controlled by the command bit PP_AUTO_LF has been connected to the PP_SELECT_IN status bit. Setting and resetting the command bit can then be observed in the status register. init_module calls this routine and sets its *run mode* to ICTP_HW if the adapter board has been found or to ICTP_SIMULATION if not.

```
static int ictp_detect(void)
{
  unsigned char value,test_bit1, test_bit2;
  test_bit1 = inb(PP_STATUS) & PP_SELECT_IN;
  printk(KERN_DEBUG "PP select status bit: %x\n",test_bit1);
  value = inb(PP_CMD);
  /* toggle the bit */
  value ^= PP_AUTO_LF;         /* ^ is the exclusive or operator! */
  outb(value,PP_CMD);

  test_bit2 = inb(PP_STATUS) & PP_SELECT_IN;
  printk(KERN_DEBUG "PP select status bit: %x\n",test_bit2);

  value ^= PP_AUTO_LF;
  outb(value,PP_CMD);
  if (test_bit1 != test_bit2)
    return TRUE;
  else
    return FALSE;
}
```

It is however possible to switch from hardware to simulation mode also under program using an ioctl (see later) call.

1.8 The Driver Routines

Since we now

- understand the hardware
- know how to install the device driver into the system
- have a frame of the driver ready
- are able to produce debugging messages

we can actually start to implement the first driver routines that do the real work. The first routines to be implemented are of course the *open* and *close* calls.

The open call usually initialises the interface. The GPI board needed programming of the 8255 chip, mainly initialisation of handshake modes and programming ports for use as input or output ports. The parallel port does not necessicate such initialisation and we skip this step.

While in its previous versions the driver restricted its use to a single program and return an EBUSY error if a second user tried to open it, the parallel port version of the driver allows multiple users. For each user the use count is incremented with MOD_INC_USE_COUNT while MOD_DEC_USE_COUNT decrements this count when closing the device. The use count is needed for dynamic unloading of the driver which can only be done if the driver is not in use by anybody anymore. Internally we also keep a use count because the driver reserves the I/O port corresponding to the parallel port with

```
request_region(io_port,3,"ictp");
```

This reservation can be seen in /proc/ioports:

```
cat /proc/ioports
0000-001f : dma1
0020-003f : pic1
0040-005f : timer
0060-006f : keyboard
0070-007f : rtc
0080-008f : dma page reg
00a0-00bf : pic2
00c0-00df : dma2
00f0-00ff : fpu
0170-0177 : ide1
01f0-01f7 : ide0
02f8-02ff : serial(auto)
0330-0333 : MPU-401 UART
0376-0376 : ide1
0378-037a : ictp                    <=======
03c0-03df : vga+
03f6-03f6 : ide0
03f8-03ff : serial(auto)
```

```
0cf8-0cff : PCI conf1
a400-a4ff : Realtek Semiconductor Co., Ltd. RTL-8139
b000-b003 : VIA Technologies, Inc. AC97 Audio Controller
  b000-b003 : via82cxxx_audio
b400-b403 : VIA Technologies, Inc. AC97 Audio Controller
  b400-b403 : via82cxxx_audio
b800-b8ff : VIA Technologies, Inc. AC97 Audio Controller
  b800-b8ff : via82cxxx_audio
d000-d01f : VIA Technologies, Inc. UHCI USB (#2)
  d000-d01f : usb-uhci
d400-d41f : VIA Technologies, Inc. UHCI USB
  d400-d41f : usb-uhci
d800-d80f : VIA Technologies, Inc. Bus Master IDE
  d800-d807 : ide0
  d808-d80f : ide1
e400-e4ff : VIA Technologies, Inc. VT82C686 [Apollo Super ACPI]
e800-e80f : VIA Technologies, Inc. VT82C686 [Apollo Super ACPI]
```

Here is the code of the open routine:

```
static int
ictp_open(struct inode * inode, struct file * file)
{
unsigned int      minor = MINOR(inode->i_rdev);
        int               ret_code;

ret_code = 0;
if (minor >= ICTP_NO)
return -ENODEV;

        ictp_write_mode  = ICTP_MODE_RAW;

/*
  allocate the I/O range corresponding to the parallel port
*/

if (!ictp_use_count) {
  if ((ret_code = check_region(io_port,3)) < 0)
    return ret_code;
  /* if (ictp_probe_hw(io_port) != 0)
return -ENODEV; */
  printk(KERN_DEBUG
            "requesting io region %3x - %3x\n",io_port,io_port+3);
  request_region(io_port,3,"ictp");
}
/*
  now setup the internal state depending on minor number
*/
        printk(KERN_DEBUG "minor: %d\n",minor);
        switch (minor) {

/*
  when reading, return current display value
*/
case ICTP_READ_DISPLAY:
  printk(KERN_DEBUG "ictp: open in 'read display mode'\n");
  break;
/*
  this allows interrupts on the push button
*/
        case ICTP_READ_IRQ_COUNT:
          ret_code = request_irq(irq,(void *)ictp_irq_interrupt,
                                   SA_INTERRUPT,"ictp",NULL);
          if (ret_code) {
              printk(KERN_WARNING "ictp: unable to use interupt 7\n");
            return ret_code;
  }
          else {
#ifdef ICTP_DEBUG
            printk(KERN_DEBUG "ictp: irq registered\n");
#endif
  }
  break;

/* some more different minor numbers */

default:
  return -EINVAL;
}

MOD_INC_USE_COUNT;
ictp_use_count++;
   return 0;
}
```

The above code is a simplified version of the code actually in service for the ictp driver. In the real *open* routine we also switch off the buzzer and, depending on the minor mode (ictp0, ictp1, ictp2) we register interrupt service routines with the system.

In order to implement the write part of the driver we should first have a look at the library routines accessible to the device driver writer. Some of these routines we have already seen before, namely:

- `register_chrdev(unsigned int major,`
  ```
                        const char *name,
                struct file_operations *fops)
  ```
- `unregister_chrdev(unsigned int major,`
  `const char *name)`
- `printk(fmt)`

There are also 2 Macros that allow us to find out the current major and minor numbers:

- `MAJOR(inode -> i_rdev) and`
- `MINOR(inode -> i_rdev)`

As we have seen in the example code above, inode is a structure that is passed into the driver routines. In order to implement the read and write routines we need additional calls to transfer a data buffer from user space to supervisory space and back. This feature is provided by:

- get_user(char *address)
- put_user(char *address)
- copy_to_user(char *user_buf, char *driver_buf, int count)
- copy_from_user(char *driver_buf, char *user_buf, int count)

Their use is demonstrated by the (incomplete) ictp write routine:

```
/*
 * Write requests on the ictp device.
 */

static int
ictp_write(struct inode * inode, struct file * file,
   const char * buf, int count)
{

        char          c;
const char    *temp=buf;
unsigned char ctemp, digit;

        switch (ictp_write_mode) {
        case ICTP_MODE_RAW:
             temp = buf;
     while (count > 0) {
```

```
        c = get_user(temp);
        outb(c,ICTP_A);
        count--;
        temp++;
      }

      return temp-buf;
      break;
```

We have now seen the *open*, *close*, *write* routines (the *read* is very similar to the *write* once we replace get_user by put_user). The only missing code is *ioctl*.

As a typical example we will have a look at the code that drives the buzzer. The buzzer can be connected to the output bit PP_INIT_PRINT that can be programmed through the parallel port's command register. The ioctl call as seen from the driver user's point of view has got 3 parameters:

- the file descriptor
- a command code
- and a parameter

Command codes and symbolic names for possible parameters are described in the ictp.h file

```
#define ICTP_SET_BUZZER IOC_OUT | 0x 0004
ioctl(ictp_fd,ICTP_SET_BUZZER,ICTP_NOISE)
ioctl(ictp_fd,ICTP_SET_BUZZER,ICTP_SILENCE)
```

will switch the buzzer on and off.

The command parameter in the driver code receives the ioctl command code while the arg is passed the corresponding argument (buzzer on or off).

```
static int
ictp_ioctl(struct inode * inode, struct file * file,
   unsigned int cmd, unsigned long arg)
{
   unsigned int minor = MINOR(inode->i_rdev);
   unsigned short dummy;

   switch (cmd) {
   case ICTP_SET_BUZZER:
#ifdef ICTP_DEBUG
     printk(KERN_DEBUG "ictp: ioctl set buzzer function entered!\n");
#endif
     if (arg == ICTP_BUZZER_ON) {
       dummy = inb(PP_CMD);
       dummy |= ICTP_BUZZER_BIT;
       outb(dummy,PP_CMD);
       return 0;
     }
     else if (arg == ICTP_BUZZER_OFF) {
       dummy = inb(PP_CMD);
       dummy &= ~ICTP_BUZZER_BIT;
       outb(dummy,PP_CMD);
       return 0;
     }
     else
       return -EINVAL;
     break;

   case ICTP_GET_BUZZER:
#ifdef ICTP_DEBUG
     printk(KERN_DEBUG "ictp: ioctl get buzzer function entered!\n");
#endif
     dummy = inb(PP_CMD);
     if (dummy & ICTP_BUZZER_BIT)
       return ICTP_BUZZER_ON;
     else
       return ICTP_BUZZER_OFF;
     break;
```

The driver allows users to choose a write mode forcing subsequent write calls to be interpreted in a different way:

- **ICTP_MODE_RAW** will send the data transferred in the write data buffer as it is to the hardware. In this mode the driver user is responsible to set up the data and chip selects correctly. This mode has been provided in order to give you trouble... well, in order to teach you the hardware.
- **ICTP_MODE_SINGLE_DIGIT** will write a single digit. Here a single byte containing the digit number (0-3) in the high nibble and the data value in the low nibble must be given in the databuffer.
- **ICTP_MODE_FULL_NUMBER** takes a short containing the number to be displayed on all for digits. This is of course the simplest way to use the driver. (Remember to specify a size of 2 (bytes) in the write call).

The full ioctl code not only permits the user to set the write mode using the ICTP_SET_WRITE_MODE ioctl command, several other commands are implemented for:

- Enabling/Disabling interrupts
- Setting the interrupt type (read becomes blocking or non blocking)
- Reading/Writing the buzzer state
- Switching run mode from simulation to hardware mode and back

1.9 Interrupts

Several times it was already mentionned that one of the reasons for writing a device driver is the possibility of treating interrupts. On the Colombo board there are several "devices" being able to produce interrupts:

- the 50/100/200 Hz clock
- the voltage to frequency converter
- the 2 pushbutton switches

One of these devices must therefore be connected to an interrupt line. As always I opted for the simplest case and I connected a pushbutton switch. When a printer is used on the parallel port the sequence of actions for a data transfer from the computer to the printer goes as follows:

- The computer puts a byte to the datalines of the parallel port (writes it to the data register).
- Then the strobe signal is toggled, telling the printer that data are available
- The printer signals back on the acknowledge line that the data have been accepted and that the next data byte can be sent.

It is therefore the parallel port's acknowledge line that is capable of generating interrupts. Before interrupts can be treated however several steps must be taken:

- An interrupt service routine, serving the incoming interrupts, must be written. It is usually this routine that clears the interrupt source (not needed in the case of the parallel port).
- The interrupt service routine must be registered with the kernel. This also enables interrupts on the CPU level.
- Interrupts must be enabled on the parallel port.

How is all this implemented in the driver? When opening the driver with minor number ICTP_READ_DISPLAY the current display value is returned if the driver is in simulation mode. If opened with minor number ICTP_READ_SWITCHES switch reading is enabled and minor number ICTP_READ_IRQ_COUNT provides access to the interrupt features.

Each time an interrupt occurs the interrupt service routines increments a counter. It is the value of this counter that is returned when reading with minor number ICTP_READ_IRQ_COUNT. After reading the counter is reset.

Before looking at the driver code let us write a little program that reads the interrupt count. You can see that the device is opened with minor number 2 (corresponding to ICTP_READ_IRQ_COUNT in the icth.h file). How would you improve the program below by using the symbolic name for the minor number?). Then an ioctl call is performed enabling interrupts on the parallel port level. After this we read the number of button presses during the last second.

```
/*
 * a test program for the ICTP device driver testing the
 * drivers interrupt capabilities
 * U. Raich 6.10.02
 *
 */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <asm/ioctl.h>
#include "ictp.h"

#define ictp_dev_name  "/dev/ictp2"
int main(int argc, char **argv)
{
  int  ictp_fd;
  int  i,ret_code;
  unsigned char count = 0;
  short current_display;

  /* try to open the device */

  ictp_fd = open(ictp_dev_name,O_RDWR);
  if (ictp_fd < 0)
    perror("irqtest: Could not open ictp device");
  else
    printf("irqtest: ictp device successfully opened!\n");

  ret_code = ioctl(ictp_fd,ICTP_ENABLE_INTERRUPT);
  if (ret_code < 0)
    perror("irqtest: Interrupt enable failed");

  /*
    this one we will see a little later
   */

  ret_code = ioctl(ictp_fd,ICTP_SET_READ_MODE,ICTP_MODE_NON_BLOCKING);
  if (ret_code < 0)
    perror("irqtest: Unable to set read mode to non blocking");

  for(i=0;i<100;i++)
    {
      ret_code = read(ictp_fd,&count,1);
      if (ret_code < 0)
```

```
        {
          perror("irqtest: Unable to read interrupt count");
          close(ictp_fd);
          exit(-1);
        }
      else
        {
          printf("Interrupt count: %d\n",count);
          sleep(1);
        }
    }
  close(ictp_fd);
}
```

When opening the ictp device with minor number ICTP_READ_IRQ_COUNT the driver registers an interrupt service routine:

```
/*
  this allows interrupts on the push button
*/
        case ICTP_READ_IRQ_COUNT:
          ret_code = request_irq(irq,(void *)ictp_irq_interrupt,
                                 SA_INTERRUPT,"ictp",NULL);
          if (ret_code) {
              printk(KERN_WARNING "ictp: unable to use interupt 7\n");
            return ret_code;
  }
          else {
#ifdef ICTP_DEBUG
            printk(KERN_DEBUG "ictp: irq registered\n");
#endif
  }
  break;
```

Again the registration of the interrupt can be seen in /proc:

```
cat /proc/interrupts
           CPU0
  0:    1409037          XT-PIC  timer
  1:      26292          XT-PIC  keyboard
  2:          0          XT-PIC  cascade
  7:          1          XT-PIC  ictp           <=====
  8:          1          XT-PIC  rtc
  9:          0          XT-PIC  usb-uhci, usb-uhci, via82cxxx
```

```
 12:     233498         XT-PIC  PS/2 Mouse
 14:     144511         XT-PIC  ide0
 15:         83         XT-PIC  ide1
NMI:          0
LOC:          0
ERR:          0
MIS:          0
```

The "ictp_irq_interrupt" routine is extremely simple: it just increments the counter.

```
/*
  first the tough part: the interrupt code
*/

static void
ictp_irq_interrupt( int irq)
{

  irq_count++;
  /*
    this we will see in a second
   */
  if (ictp_read_mode == ICTP_MODE_BLOCKING)
    wake_up(&ictp_wait_q);
}
```

All that is still missing is the ioctl code for enabling interrupts on the parallel port. This is done by setting the PP_EN_IRQ bit in the parallel port command register.

```
    case ICTP_ENABLE_INTERRUPT:
#ifdef ICTP_DEBUG
     printk(KERN_DEBUG "ictp: enabling interrupts on parallel port\n");
#endif

     if (minor == ICTP_READ_IRQ_COUNT) {
       dummy = inb(PP_CMD);
       dummy |= PP_EN_IRQ;
       outb(dummy,PP_CMD);
       irq_count = 0;
```

Even though we are now able to treat interrupts we are still not able to synchronise a user program to an external event. Synchronisation however is of great interest when implementing control systems. Often we must wait for some external hardware before the control sequence can be continued. Synchronisation is accomplished by a *blocking read*. Instead of returning immediately giving the current irq count, which may well be zero, the read function will now block until an interrupt comes in. We have seen in the example above that *nonblocking reads* were used. Synchronisation can be reached when we switch the drivers interrupt mode to *blocking*.

```
/*
 * a test program for the ICTP device driver testing the
 * drivers interrupt capabilities
 * U. Raich 6.10.02
 *
 */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <asm/ioctl.h>
#include "ictp.h"

#define ictp_dev_name  "/dev/ictp2"
int main(int argc, char **argv)
{
  int  ictp_fd;
  int  i,ret_code;
  unsigned char count = 0;
  short current_display;

  /* try to open the device */

  ictp_fd = open(ictp_dev_name,O_RDWR);
  if (ictp_fd < 0)
    perror("irqtest: Could not open ictp device");
  else
    printf("irqtest: ictp device successfully opened!\n");

  ret_code = ioctl(ictp_fd,ICTP_ENABLE_INTERRUPT);
  if (ret_code < 0)
    perror("irqtest: Interrupt enable failed");

  ret_code = ioctl(ictp_fd,ICTP_SET_READ_MODE,ICTP_MODE_BLOCKING);
  if (ret_code < 0)
    perror("irqtest: Interrupt enable failed");

  for(i=0;i<100;i++)
    {
      printf("Falling asleep. rrrrrrr.....\n");
      ret_code = read(ictp_fd,&count,1);
      printf("Oups, I woke up due to interrupt!\n");
      sleep(1);
    }
  close(ictp_fd);
}
```

In order to implement this feature in the driver we need a wait queue into which all users of the driver waiting for an event will be entered. When the user program issues a read call it will fall asleep until and it will be awaken by the driver as soon as an interrupt has occured.

The wait queue is defined and initialised by a support macro

DECLARE_WAIT_QUEUE_HEAD (ictp_wait_q);

and here is the piece of the read routine that sleeps until it is awaken by the interrupt service routine.

```
case ICTP_READ_IRQ_COUNT:
#ifdef ICTP_DEBUG
          printk(KERN_DEBUG "ictp: irq_count: %d\n",irq_count);
#endif
  if (count != 1)
    return -EINVAL;
          if (ictp_read_mode == ICTP_MODE_BLOCKING){
            if (irq_count == 0) {
#ifdef ICTP_DEBUG
      printk(KERN_DEBUG "ictp: Going to sleep ...\n");
#endif
              interruptible_sleep_on(&ictp_wait_q);
    }
```

1.10 Appendix A: The ICTP device driver user's manual

A sample device driver for the ICTP board has been developed. The following gives a summary of its functions.

The ictp driver expects a Colombo board connected to the PC parallel port via an adapter as described in fig. 1.3. Any standard parallel port can be used. By default lp1 with I/O address 0x378 is used but another I/O adress can be specified at module load time giving the option "base=" address (e.g. 0x278) to insmod. The connections to the ICTP board must be made as follows:

- Data Port when writing: ICTP displays
  Data Port when reading (with major number 1)
- The Colombo board signal intended for interrupt generation must be connected to the parallel port's acknowledge line.

The driver functions:

Read calls:

The driver uses dynamic major number allocation (usually major = 254) and 2 minor numbers:

- read on minor number *ICTP_READ_SWITCHES*:
  - read the current switch settings
- read on minor number *ICTP_READ_IRQ_COUNT:*
  - returns the number of
  - interrupts arrived since
  - the last read call.

Reads for interrupts exist in 2 flavors:

- **non blocking**: The number of interrupts since the last read is immediately returned, even if it is zero.
- **blocking**: If the number of interrupts is zero, it blocks the calling process until the next interrupt (or other signal like `^C`) arrives.

Write calls:

Writing works on any of the three minor devices. There are 3 different write modes which may be set up by *ioctl* calls (see later).

- **ICTP_MODE_RAW**: in this mode the data coming from the user are sent untreated to the I/O port. In order to make the displays work correctly, the user must select the suitable data/chipselect sequences cs high + data, cs low + data, cs high + data for all digits. In each byte the high nibble specifies the data and the low nibble the chip selects. 12 data bytes are expected and the driver will return **EINVAL** if the count is wrong
- **ICTP_MODE_SINGLE_DIGIT:** a single data byte is accepted. The high nibble contains the digit number (0-3) and the low nibble contains the data.
- **ICTP_MODE_FULL_NUMBER**: a short is expected. This number will be put onto the digits.

ioctl calls:

•	ICTP_SET_WRITE_MODE:	sets up the writing mode. The following values are accepted:
—	ICTP_MODE_RAW	12 data bytes
—	ICTP_MODE_SINGLE_DIGIT	only 1 data byte allowed
—	ICTP_MODE_FULL_NUMBER	a short needed;
•	ICTP_SET_READ_MODE	set the read mode
—	ICTP_MODE_BLOCKING	if count = zero, block process until interrupt arrives
—	ICTP_MODE_NON_BLOCKING	return current count immediately
•	ICTP_SET_RUN_MODE	set the run mode
—	ICTP_HW	run in normal mode accessing the Colombo board
—	ICTP_SIMULATION	run a *virtual* Colombo board, allows implementation of a simulator
•	ICTP_GET_WRITE_MODE	return the current write mode
•	ICTP_GET_READ_MODE	return the current read mode
•	ICTP_GET_RUN_MODE	return the current run mode
•	ICTP_SET_SWITCHES	argument: new simulated switch settings
•	ICTP_GET_DISPLAY	returns the current display setting
•	ICTP_ENABLE_INTERRUPT	enable interrupts on parallel port level
•	ICTP_DISABLE_INTERRUPT	disable interrupts on parallel port level
•	ICTP_SET_BUZZER:	Buzzer control. The following values are accepted:
—	ICTP_SILENCE	switches buzzer off
—	ICTP_NOISE	switches buzzer on

1.11 Appendix B: The ICTP device driver listings

Since the include file is shown in its complete form in the chapter on driver implementation and the driver code only in code snippets, we print here the entire listing of the driver code.

```
/*
 * Implements the ICTP character device driver.
 * Create the device with:
 *
 * mknod /dev/ictp c 254 0
 *
 * - U. Raich
 * 13.03.94 : First version working with PC parallel printer port
 *
 * Modifications:
 * 30.08.94 : U.R. complete rewrite for Manuel's board
 *
 * 06.10.02 : U.R. coming back to a PC parallel printer port version
 */

/* Kernel includes */

#include <linux/module.h>
#include <linux/init.h>

#include <linux/config.h>
#include <linux/errno.h>
#include <linux/kernel.h>
#include <linux/major.h>
#include <linux/sched.h>
#include <linux/slab.h>
#include <linux/fcntl.h>
#include <linux/delay.h>
#include <linux/ioport.h>
#include "ictp.h"

#include <asm/irq.h>
#include <asm/io.h>
#include <asm/uaccess.h>
#include <asm/system.h>

#define ICTP_DEBUG   1
```

```
/*
   some globals:
*/
int                 ictp_major = ICTP_MAJOR;
int                 io_port    = ICTP_IO_PORT;
int                 irq        = ICTP_IRQ;
int                 ictp_use_count = 0;

MODULE_PARM(ictp_major,"i");
MODULE_PARM_DESC(ictp_major,"ICTP’s major number");
MODULE_PARM(io_port, "i");
MODULE_PARM_DESC(io_port, "Base address of parallel I/O card");
MODULE_PARM(irq,"i");
MODULE_PARM_DESC(irq,"ICTP’s interrupt number");

MODULE_AUTHOR("Ulrich Raich");
MODULE_LICENSE("GPL");

unsigned long     ictp_write_mode  = -1;
unsigned long     ictp_read_mode   = ICTP_MODE_NON_BLOCKING;
short             current_display=0, old_current_display  = 0;
unsigned char     sim_switch_settings = 0;
unsigned char     run_mode = ICTP_HW;
unsigned char     irq_count = 0;

DECLARE_WAIT_QUEUE_HEAD (ictp_wait_q);

/*
 * The driver.
 */

static int ictp_detect(void)
{
  unsigned char value,test_bit1, test_bit2;
  test_bit1 = inb(PP_STATUS) & PP_SELECT_IN;
  printk(KERN_DEBUG "PP select status bit: %x\n",test_bit1);
  value = inb(PP_CMD);
  /* toggle the bit */
  value ˆ= PP_AUTO_LF;
  outb(value,PP_CMD);

  test_bit2 = inb(PP_STATUS) & PP_SELECT_IN;
  printk(KERN_DEBUG "PP select status bit: %x\n",test_bit2);
  value ^= PP_AUTO_LF;
  outb(value,PP_CMD);
  if (test_bit1 != test_bit2)
    return TRUE;
  else
    return FALSE;
}


static void
out_digit(unsigned char digit, unsigned char number)

{
  unsigned char mask,c;
  unsigned short short_mask;
  unsigned char pp_cmd;

  pp_cmd = PP_WRITE;        /* set parallel port to "write" mode */
  outb(pp_cmd,PP_CMD);

  mask = 1 << digit;
  mask = ~mask;
#ifdef ICTP_DEBUG
  printk(KERN_DEBUG "ictp: mask %x\n",mask);
#endif
  c = (number << 4) | 0xf;
  outb(c,PP_DATA);
  c &= mask;
  outb(c,PP_DATA);
  c |= 0xf;
  outb(c,PP_DATA);
  /* save this digit in current_display */

#ifdef ICTP_DEBUG
    printk(KERN_DEBUG
           "ictp: digit: %d, number: %d\n",digit,number);
#endif
  short_mask = 0xf;
  short_mask <<=  4*digit;
  current_display &= ~short_mask;   /* take out old value */
  current_display |= (number << 4*digit);
}
```

```
/*
  first the tough part: the interrupt code
*/

static void
ictp_irq_interrupt( int irq)
{

  irq_count++;
  if (ictp_read_mode == ICTP_MODE_BLOCKING)
    wake_up(&ictp_wait_q);
}

/*
 * Handle ioctl calls
 */

static int
ictp_ioctl(struct inode * inode, struct file * file,
   unsigned int cmd, unsigned long arg)
{
   unsigned int minor = MINOR(inode->i_rdev);
   unsigned short dummy;

#ifdef ICTP_DEBUG
    printk(KERN_DEBUG
      "ictp: ioctl write function entered! cmd: %x, arg: %lx\n",
      cmd, arg);
#endif
   switch (cmd) {

   case ICTP_SET_BUZZER:
#ifdef ICTP_DEBUG
     printk(KERN_DEBUG
            "ictp: ioctl set buzzer function entered!\n");
#endif
     if (arg == ICTP_BUZZER_ON) {
       dummy = inb(PP_CMD);
       dummy |= ICTP_BUZZER_BIT;
       outb(dummy,PP_CMD);
       return 0;
     }
     else if (arg == ICTP_BUZZER_OFF) {
       dummy = inb(PP_CMD);
       dummy &= ~ICTP_BUZZER_BIT;
       outb(dummy,PP_CMD);
       return 0;
     }
     else
       return -EINVAL;
     break;

   case ICTP_GET_BUZZER:
#ifdef ICTP_DEBUG
     printk(KERN_DEBUG
            "ictp: ioctl get buzzer function entered!\n");
#endif
     dummy = inb(PP_CMD);
     if (dummy & ICTP_BUZZER_BIT)
       return ICTP_BUZZER_ON;
     else
       return ICTP_BUZZER_OFF;
     break;

   case ICTP_SET_WRITE_MODE:
    if (arg > ICTP_MODE_SWITCHES)
 return -EINVAL;
     else {
         ictp_write_mode = arg;
 return 0;
       }
     break;

   case ICTP_GET_DISPLAY:
#ifdef ICTP_DEBUG
     printk(KERN_DEBUG
       "ictp: ioctl get display function entered!\n");
#endif
     return current_display;
     break;

   case ICTP_SET_SWITCHES:
     sim_switch_settings = (unsigned char) arg & 0xff;
 return 0;

   case ICTP_GET_WRITE_MODE:
#ifdef ICTP_DEBUG
     printk(KERN_DEBUG
        "ictp: ioctl read function entered! cmd: %x\n",cmd);
#endif
     return ictp_write_mode;
     break;

   case ICTP_SET_RUN_MODE:
#ifdef ICTP_DEBUG
    printk(KERN_DEBUG
       "ictp: ioctl write function entered! cmd: %x, arg: %lx\n",
cmd, arg);
#endif
    if (arg > ICTP_HW)
 return -EINVAL;
     else {
         run_mode = arg;
 return 0;
       }
     break;

   case ICTP_GET_RUN_MODE:
#ifdef ICTP_DEBUG
     printk(KERN_DEBUG
       "ictp: ioctl read function entered! cmd: %x\n",cmd);
#endif
     return run_mode;
     break;

   case ICTP_SET_READ_MODE:
#ifdef ICTP_DEBUG
    printk(KERN_DEBUG
       "ictp: ioctl write function entered! cmd: %x, arg: %lx\n",
cmd, arg);
#endif
    if (arg > ICTP_MODE_NON_BLOCKING)
 return -EINVAL;
     else {
         ictp_read_mode = arg;
 return 0;
       }
     break;

   case ICTP_GET_READ_MODE:
#ifdef ICTP_DEBUG
     printk(KERN_DEBUG
       "ictp: ioctl read function entered! cmd: %x\n",cmd);
#endif
     return ictp_read_mode;
     break;

    case ICTP_ENABLE_INTERRUPT:
#ifdef ICTP_DEBUG
     printk(KERN_DEBUG
       "ictp: enabling interrupts on parallel port\n");
#endif

     if (minor == ICTP_READ_IRQ_COUNT) {
       dummy = inb(PP_CMD);
       dummy |= PP_EN_IRQ;
       outb(dummy,PP_CMD);
       irq_count = 0;

#ifdef ICTP_DEBUG
       printk(KERN_DEBUG "ictp: interrupts are enabled\n");
#endif
       return 1;
     }
     else
       return -EINVAL;
     break;

   case ICTP_DISABLE_INTERRUPT:
#ifdef ICTP_DEBUG
     printk(KERN_DEBUG "ictp: disabling interrupts\n");
#endif
     if (minor == ICTP_READ_IRQ_COUNT) {
       dummy = inb(PP_CMD);
       dummy &= ~PP_EN_IRQ;
       outb(dummy,PP_CMD);
       return 1;
     }
     else
       return -EINVAL;
     break;

   default:
     return -EINVAL;
   }
}
```

```
/*
 * Read the status of the ICTP board switches
 */
static ssize_t
ictp_read(struct file * file, char * buf, size_t count,
                                           loff_t *ppos)
{

unsigned int  minor = MINOR(file->f_dentry->d_inode->i_rdev);
        unsigned char testvalue;
unsigned char pp_cmd;

        switch (minor) {
case ICTP_READ_SWITCHES:
  if (count != 1)
    return -EINVAL;
  /*
     read the switches if not in simulation mode
  */
  if (run_mode != ICTP_SIMULATION)
    {
      pp_cmd    = PP_BI_DIRECT;
      outb(pp_cmd,PP_CMD);

      testvalue = inb(PP_DATA);        /* read the switches */
#ifdef ICTP_DEBUG
      printk(KERN_DEBUG
     "ictp: switch value read from port: %x\n",
                       testvalue);
#endif
    }
  else
    {
      testvalue = sim_switch_settings;
#ifdef ICTP_DEBUG
      printk(KERN_DEBUG
     "ictp: simulated switch value: %x\n",testvalue);
#endif
    }
  if (put_user(testvalue,buf))
    return -EFAULT;
  else
    return 1;
          break;
```

```
case ICTP_READ_IRQ_COUNT:
#ifdef ICTP_DEBUG
          printk(KERN_DEBUG "ictp: irq_count: %d\n",irq_count);
#endif
  if (count != 1)
    return -EINVAL;
          if (ictp_read_mode == ICTP_MODE_BLOCKING){
            if (irq_count == 0) {
#ifdef ICTP_DEBUG
      printk(KERN_DEBUG "ictp: Going to sleep ...\n");
#endif
              interruptible_sleep_on(&ictp_wait_q);
    }
#ifdef ICTP_DEBUG
            printk(KERN_DEBUG "ictp: returned from sleep\n");
#endif
            if (put_user(irq_count,buf))
      return -EFAULT;
            irq_count = 0;
          }
          else {
            if (put_user(irq_count,buf))
      return -EFAULT;
            irq_count = 0;
          }
          return 1;
default:
  return -EINVAL;
}
}
/*
 * Write requests on the ictp device.
 */
static ssize_t
ictp_write(struct file * file, const char * buf, size_t count,
                                                  loff_t *ppos)
{

        char          c;
const char    *temp=buf;
unsigned char ctemp, digit, pp_cmd;

        switch (ictp_write_mode) {
```

```
        case ICTP_MODE_RAW:
#ifdef ICTP_DEBUG
  printk(KERN_DEBUG "writing in raw mode\n");
#endif

  pp_cmd = PP_WRITE;    /* set parallel port to "write" mode */
  outb(pp_cmd,PP_CMD);

  if (count != 12)
    return -EINVAL;
  old_current_display = current_display;
  current_display = 0;
  temp = buf;
  while (count > 0) {
    if (get_user(c,(char *)temp))
      {
printk(KERN_ERR "could not get user buffer\n");
return -EFAULT;
      }
    outb(c,PP_DATA);

    if ((count % 3) == 2)
      {
switch (c&0xf)
  {
  case 0x0e:
    current_display |= (c >> 4);
    break;
  case 0x0d:
    current_display |= (c & 0xf0);
    break;
  case 0x0b:
    current_display |= (((short)c & 0xf0) << 4);
    break;
  case 0x07:
    current_display |= (((short)c & 0xf0) << 8);
    break;
  default:
    current_display = old_current_display;
    printk(KERN_ERR "Invalid strobe nibble: %x\n",c&0xf);
    return -EFAULT;
  }
      }
    count--;
    temp++;
  }


  return temp-buf;
  break;

        case ICTP_MODE_SINGLE_DIGIT:
  if (count != 1)
    return -EINVAL;
  if (get_user(c,(char *)temp))
    return -EFAULT;
  digit = c >> 4;
  ctemp = c & 0xf;
  out_digit(digit,ctemp);
  return 1;
  break;
case ICTP_MODE_FULL_NUMBER:
  if (count != 2)
    return -EINVAL;
  temp = buf;
  if (get_user(c,(char *)temp))
    return -EFAULT;
#ifdef ICTP_DEBUG
  printk(KERN_DEBUG "write, mode 2, first byte: %x\n",c);
#endif
  ctemp = c & 0xf;
  out_digit(0,ctemp);
  ctemp = c >> 4;
  out_digit(1,ctemp);

  if (get_user(c,(char *)temp+1))
    return -EFAULT;
#ifdef ICTP_DEBUG
  printk("write, mode 2, second byte: %x\n",c);
#endif
  ctemp = c & 0xf;
  out_digit(2,ctemp);
  ctemp = c >> 4;
  out_digit(3,ctemp);
/*
  get first nibble
*/
```

```
  return 2;
  break;
case ICTP_MODE_SWITCHES:
  if (run_mode)
    return -EINVAL;
  if (count != 1)
    return -EINVAL;
  if (get_user(c,(char *)temp))
    return -EFAULT;
  sim_switch_settings = c;
  return 1;
  break;

default:
  return 1;
  break;
}
}

static int
ictp_open(struct inode * inode, struct file * file)
{
unsigned int      minor = MINOR(inode->i_rdev);
        int               ret_code;

ret_code = 0;
if (minor >= ICTP_NO)
return -ENODEV;

        if (ictp_write_mode == -1)
  ictp_write_mode = ICTP_MODE_RAW;

/*
  allocate the I/O range corresponding to the parallel port
*/
if (!ictp_use_count) {
  if ((ret_code = check_region(io_port,3)) < 0)
    return ret_code;
  /* if (ictp_probe_hw(io_port) != 0)
return -ENODEV; */
  printk(KERN_DEBUG
                 "requesting io region %3x - %3x\n",io_port,io_port+3);
  request_region(io_port,3,"ictp");
}
```

```
/*
  now setup the internal state depending on minor number
*/
        printk(KERN_DEBUG "minor: %d\n",minor);
        switch (minor) {

/*
  this allows interrupts on the push button
*/
        case ICTP_READ_IRQ_COUNT:
          ret_code = request_irq(irq,(void *)ictp_irq_interrupt,
                                 SA_INTERRUPT,"ictp",NULL);
          if (ret_code) {
              printk(KERN_WARNING "ictp: unable to use interupt 7\n");
            return ret_code;
  }
          else {
#ifdef ICTP_DEBUG
            printk(KERN_DEBUG "ictp: irq registered\n");
#endif
  }
  break;

case ICTP_READ_SWITCHES:

/*
  kill the buzzer
  first setup port C to bit set (bit set/reset mode with set bit on! )
*/


#ifdef ICTP_DEBUG
          printk(KERN_DEBUG "ictp: opened for switch reading\n");
#endif
          break;

default:
  return -EINVAL;
}

MOD_INC_USE_COUNT;
ictp_use_count++;
   return 0;
```

```
}

static int
ictp_release(struct inode * inode, struct file * file)
{
unsigned int minor = MINOR(inode->i_rdev);

/*
  free the interrupt
*/
if (minor == ICTP_READ_IRQ_COUNT)
  {
    free_irq(irq,NULL);
#ifdef ICTP_DEBUG
    printk(KERN_DEBUG "ictp: interrupt free’d\n");
#endif
  }

if (ictp_use_count == 1) {
  printk(KERN_DEBUG
             "releasing io region %3x - %3x\n",io_port,io_port+3);
  release_region(io_port,3);
}

ictp_use_count --;
MOD_DEC_USE_COUNT;
return 0;
}

static struct file_operations ictp_fops = {
  read:     ictp_read,
  write:    ictp_write,
  ioctl:    ictp_ioctl,
  open:     ictp_open,
  release:  ictp_release,
};

/*
 * And now the modules code and kernel interface.
 */

int
init_module( void) {
  int result;
```

```
#ifdef ICTP_DEBUG
  printk(KERN_DEBUG "ictp:  init_module called\n");
  printk(KERN_DEBUG "ictp:  io base address %x\n",io_port);
#endif

  /*
    register the device driver with the system
  */
  result=register_chrdev(ictp_major, "ictp", &ictp_fops);

  if (result < 0)
    {
      printk(KERN_ERR "register_chrdev failed: goodbye world :-(\n");
      return -EIO;
    }
  if (ictp_major == 0)  /* dynamic major number allocation */
    ictp_major=result;

#ifdef ICTP_DEBUG
printk(KERN_DEBUG "ictp: driver registered!\n");
#endif
if (ictp_detect())
  run_mode = ICTP_HW;
else
  run_mode = ICTP_SIMULATION;
  return 0;
}

void
cleanup_module( void) {

#ifdef ICTP_DEBUG
  printk(KERN_DEBUG "ictp:  cleanup_module called\n");
#endif

  if (unregister_chrdev(ictp_major, "ictp") != 0) {
    printk("cleanup_module failed\n");
  }
#ifdef ICTP_DEBUG
  else
    printk(KERN_DEBUG "cleanup_module succeeded\n");
#endif
}
```