Linux Device Drivers

Level 3 Project

Final Report

Student: Grigorios Fragkos   E/N:01066900
BSc Software Engineering

Supervisors: Dr. Gaius Mulley
Mr. Keith Verheyden

University of Glamorgan   *Wales 2002 – 2003*

Statement of Originality:

SCHOOL OF COMPUTING

DEGREE SCHEME IN COMPUTING
LEVEL THREE PROJECT

This is to certify that, except where specific reference is made, the work described within this project is the result of the investigation carried out by myself, and that neither this project, nor any part of it, has been submitted in candidature for any other award other than this being presently studied.

Any material taken from published texts or computerised sources have been fully referenced, and I fully realise the consequences of plagiarising any of these sources.

Student Name (Printed)	Grigorios Fragkos……………
Student Signature	………………………………..
Registered Scheme of Study	BSc (Hons) Software Engineering
Date of Signing	09/May/2003……………..…..

Abstract:

The title of the project is Linux Device Drivers and is about how to design and develop a device driver from scratch for an ISA I/O card in Linux. It also includes a basic introduction on Linux Operating System and analyses the various concepts of the Linux kernel. Slackware 8.1 was the Linux distribution used for this project running on the 2.4.18 official Linux kernel. The module and the user space program were written in C programming language.

This project provide as with all the basic knowledge and the opportunity to learn how to use the computer in order to control any type of hardware device. We could connect any type of device that sends (and/or) receives binary signals (electronic circuit) in order to access it from a computer. That device could be a barometer, a thermometer even an electronic door lock.

There is a large flexibility in the areas this project could be used. Except the great knowledge we obtain about the core of the Operating Systems and the ability to control hardware inside and outside a computer, we are also in place to use our imagination to discover where it would be helpful to every and each one of us.

Acknowledgements:

I would like to thank some people for their help in order to make a little bit easier to me to do my research and develop my final year project.

My supervisor Dr. Gaius Mulley for being always there in order to answer any questions I had and encouraging me about the progress of the project.

Mr. Yiannis Koukouras for helping me take my first steps as a simple Linux user and providing my with the knowledge of how I should think when I have to deal with Linux distributions.

Mr. Nikolaos Avourdiadis for letting me borrow his books about Linux and providing me with the Linux handouts he had from last year when he was a postgraduate student.

Contents:

Statement of Originality: 2

Abstract:...3

Acknowledgements: 4

Contents: 5

Introduction and Objectives:...9

Abbreviations used: 9

Linux Device Drivers – Final Report...10

Introduction:..10

Project Analysis: Time – Plan 12

Linux: 13

Familiarization with Linux: 13

Shell Scripting 20

Kernel: 21

Init runlevels 21

Recompile your kernel 22

The LILO boot loader...24

Kernel programming 26

ISA Input Output Card 27

How a module works:...30

Devices..31

Device Major Number 32

Device name...33

Usage Count..33

How can we create a node..34

Loading modules automatically..34

User Space Program:..34

How the ISA I/O card was programmed:..36

Step through the source code of the Module: 37

Writing a script for easier manipulation:..39

Interrupts:..39

The overall progress summary..41

Testing of the project..42

Screenshots – System goes live 45

Conclusions..46

References: 47

Appendix 1 – Project Time Plan..49

Appendix 2 – User Space program (source code final version) 50

Appendix 3 – Kernel Module (source code final version) 53

Appendix 4 – MS – DOS application (source code final version)..57

Appendix 5 – The ISA I/O card PCL731 59

1st Sub-Report 60

About the report: 60

Introduction:....................60

Linux: 62

Familiarization with Linux: *62*

Shell Scripting *69*

Kernel: 70

Init runlevels *70*

Recompile your kernel *71*

The LILO boot loader *73*

Kernel programming *75*

ISA Input Output Card 76

References: 79

Appendix A 80

Hello_kernel.c *80*

Appendix B 81

MyFile.sh *81*

2nd Sub-Report 82

About the report: 82

Introduction:....................82

How a module works:....................83

Devices *84*

Device Major Number *85*

Device name *86*

Usage Count......86

How can we create a node......86

Loading modules automatically......87

User Space Program:......87

How the ISA I/O card was programmed:......89

Step through the source code of the Module:......91

Writing a script for easier manipulation:......92

Interrupts:......93

References:......94

Appendix 1:......95

//user space program:......95

Appendix 2:......97

//Module Source Code......97

Progress Reports:......102

Objectives Settings Report......150

Agreed Marking Scheme Weightings......152

All References:......154

Bibliography......154

Webliography......155

EXTRAS:
(PCL-731 48 Sit Digital I/O Card User's Manual)

Introduction and Objectives:

Aim of the project is to construct a Linux Kernel module to drive an ISA input/output card. This concept for final year project was an original idea proposed by the student to the final year tutor Mr. Phil Davies.

This project gives the opportunity of researching the structure of an operating system and particularly the relationship between user-space and kernel space. Furthermore, it familiarizes the student with concepts of Linux Kernel programming. The principles of the ISA bus interface and how a software program can access hardware parts of a computer, send and receive information in order to make it available to react with the outside world should be researched.

All the research is should be combined in order to design and develop a module for the hardware device. This module will be available to be accessed by a user space program that was developed.

The original idea of researching the kernel of an operating system and starting writing source code for accessing a hardware device came up from the Operating Systems module we had in the second year of our degree scheme. Also, by the author's experience, software engineers should know how to make the computer a tool to their hands in order to control hardware devices as much as inside and outside the PC case by code.

Abbreviations used:

MP: (#) *command* = for more information refer to the manpage of *command* in section number # with the command line "man # *command*"

Every reference has a unique identifier. At end of each paragraph there is a tag similar to this [Reference a3, a4] that refers to the each relevant reference. (Page 156)

Linux Device Drivers – Final Report

Introduction:

The following report is an introduction to Linux operating system and teaches you the concepts involved with kernel programming in order to design and implement a kernel module. In this case study a device driver for an I/O ISA card will be implemented. We will start with a familiarization with Linux and we will end writing the source code for the module using the C++ programming language.

Linux is an operating system that is spreading fast into the computing market. We can consider of Linux as the rising star amongst operating systems. A major question must be answered before we continue, why people like Linux? The other operating systems have a lot of limitations, they are designed for low-end home users and do not deliver the performance or flexibility that is expected. Furthermore Linux is free, stable, and runs on the majority of processors like Alpha, Sparc, Pentium Pro, Pentium II, Pentium III, Pentium IV, AMD, Cyrix chips and even older 386/486 machines as well as some other platforms. In fact Linux can be ported for almost every platform, even for R.T.E.C.S[1]. At this point we should mention that Linux is a redistributable clone of the UNIX operating system which uses the X window system graphical user interface (GUI) and offers several different integrated desktop environments such as KDE and GNOME. Despite the fact that Linux supports a great majority of devices, a great part of the manufacturing companies, that produce computer-related hardware, do not support their products with Linux device drivers yet. Subsequently, development of those drivers is needed to be done by the users themselves.

The Kernel of the operating system is a bunch of code that acts as an interface between the application layer and the hardware. The application layer is the user space environment where all the applications are executed by the user. In order for the user-space program to communicate with the hardware from the application layer, the

[1] Real Time Embedded Computer Systems

interception of a kernel is vital. The kernel looks for the particular driver for the device we want to access and, according to the instructions on that driver, will allow the program to communicate with the specific piece of hardware (e.g. Hard Drive, CD-Rom, ISA card). This driver is a module for the kernel usually written in C programming language. Actually the most of the kernel is made up from too many kernel modules linked together.

The Peripheral Component Interconnect (PCI) bus allows us to connect a hardware card into our computer. An example of a PCI card is the well-known soundcard we have in our computers. Nowadays, the PCI standard as a result of its cost effectiveness, backward compatibility, scalability, and forward-thinking design, has overcome ISA (Industry Standard Architecture) standard which is the ancestor of the PCI bus. Even thought, the ISA bus architecture is still used by most manufacturers; this is due to the fact that the transaction from the one architecture to the other is very expensive for the most of the industry and the managers think that is this is not a value for money solution.

The object of this project is to familiarize the student with the concepts of Linux Kernel programming, the GNU/Linux Operating System, ISA bus interface and C programming, in order to develop, implement and test a kernel module which will provide the appropriate environment to a user-space application to access the ISA I/O card.

In the second part of the report we are going to cover the major aspects concerning a device module implementation. We will start approaching the kernel structure and how it handles the modules in order to access devices. Furthermore, we will see how a module links to the kernel and how it performs the various operations. Additionally, we will explain the role of the user-space program in order to access the device. In order to make use of the I/O card we will analyze how the ISA card was programmed using the specification of the user's manual. The issues concerning the C code that was produced and the source code structure will be covered.
[Reference a2, a4]

Project Analysis: Time – Plan

At the beginning of the project a time – plan was produced in order to organize the time spend to each task for the project. The time – plan used as a guideline for each task and it helped meeting the deadlines successfully. The figure found in appendix 1 shows the time – plan that produced using a Gantt chart.

The total duration for the project was 201 days, started on Monday 21/Oct/2002 and has ended on Friday 9/May/2003. (Appendix 1, Page 49)

The time – plan of the actual work of the project, as it happened during the 201 days of implementation, is almost similar to the planed work that is showed in the Gantt chart. Any differences between the plan work and the actual word took place during the implementation has been reported in the progress reports.

Any changes to the time – plan were totally agreed with the supervisor in advance.

Linux:

“*Linux is a UNIX clone written from scratch by Linus Torvalds in 1992 with assistance from a team of developers across the network.*”[2] It was first developed for 386/486-based PCs. These days it also runs in several other machines like DEC Alphas, SUN Sparcs, M68000 machines (like Atari and Amiga), MIPS and PowerPC. During this 10 years, that Linux exists, it has become a very wide-use operating system, mostly used by well experienced users and for server or embedded computers use.

However, it still can’t overcome the Windows® monopoly on the market. Therefore, most device manufacturers, for the above platforms, do not make any drivers to support their products for the Linux environment. This leaves, the Linux users, with two options; either not buy the product or build the drivers by themselves. C is a function oriented programming language, which was devised in the early 1970s as a system implementation language for the nascent UNIX operating system. Nowadays, it has become one of the dominant languages. The Linux kernel and most of its modules were built in C platform. By 1973 UNIX OS was almost totally written in C. Thus, in case that someone needs to update the kernel or implement a device driver, C language is the best approximation to do that. [Reference a2, a4]

Familiarization with Linux:

Linux has many distributions; the most popular are **Red Hat, SuSe, Slackware and Debian**. Usually you choose the right distribution for the right job. In the following pages we will explore the world of Linux through the **Slackware 8.1** distribution which uses the **2.4.18 official Linux kernel**. The first thing we must describe is how to install Slackware into our system.

[2] http://mirror.ati.com/support/faq/linux.html

Considering that the most users have Windows® installed into their system we must say that those two operating systems can co-exist into the same operation system, for that reason there is no need of uninstalling Windows® in order to install any of the distributions of Linux. The only thing we must be aware is that we should install Linux in a separate partition on our hard drive.

In that separate partition we should create two partitions; the first is the root partition where all the GNU files among the Linux kernel will be installed, the second is the swap partition (a partition which will be used as extra memory for the operating system). Some other distributions, like Red Hat®, choose to use an extra partition for the `/boot` directory, where the Linux kernel and all the booting configuration files will be stored.

Except the `fdisk` partition table manipulator, GNU/Linux uses some other similar, yet more user friendly, programs such as the "Disk Druid" and `cfdisk`, in order to help as partition our hard drive. Red Hat and Slackware have a user friendly interface in order to help us install GNU/Linux into our system. The file systems supported by GNU/Linux are plenty, like reiserfs, fat32®, minix®, ISO9660, udf, ext2/3 and MS-DOS®. (MP (8) mount)

Usually, we use the console environment as the basic environment which is something like MS-DOS® command line in Windows® but with furthermore capabilities and functionalities. The administrator login in Linux is called root and we navigate through the directories using the following commands (`cd`, `cd ..`, `cd /`, `ls`, etc). In order to find a program or command that is suitable for a specific operation, we can search through the *apropos* (MP (1) apropos) database and then use it's manpage (MP (1) man) to understand the way it works. In order to exit a manpage press the `Q` key on your keyboard.

All devices in Linux have a special file to be represented. This special files are into the `/dev` directory and the following table have some of them. In general, everything, from a device to a memory map, in GNU/Linux is represented by a file. This means, in order to access it, we have to use the normal file operations we would use for a simple text document.

Special File Form	**Example**	**Device / Use**
/dev/fd*n*	/dev/fd0	Floppy Disk
/dev/rmt*n*	/dev/rmt0	Generic tape devices
/dev/rst*n*	/dev/rst0	SCSI tape drive
/dev/cd*n*	/dev/cd0	CD-ROM devices
/dev/tty*n*	/dev/tty01	Serial Line (hardware terminal / modem)
/dev/pts/*n*	/dev/pts/0	Pseudo-terminal (used for network sessions)
/dev/console	/dev/console	Console Device
/dev/mem	/dev/mem	Map of physical memory
/dev/kmem	/dev/kmem	Map of Kernel virtual memory
/dev/mouse	/dev/mouse	Mouse Interface
/dev/ram*n*	/dev/ram00	Ram Disk
/dev/sd/a-g/*n*	/dev/sda0	SCSI Disk
/dev/swap	/dev/swap	Swap device
/dev/null	/dev/null	Null device: output written to it is discarded

(Introduction to UNIX security, Dr. Andrew Blyth, Lecture Notes 2002, MSc Information Systems and Network Security, University of Glamorgan)

A simple PS/2 mouse will work perfectly in GNU/Linux but in case we have a wheel PS/2 mouse the following hint might be very useful for the users that has get used to working with the wheel in order to scroll through the pages of a document. Suppose we have an IntelliMouse mice. In order to make the mouse wheel to work we should open the `X1186Config` file with the `jed` editor by typing the following command. (Remember that is case sensitive)

```
jed /etc/X11/XF86Config
```

We are looking for the line that into the file that has the following command.

```
Option "Protocol"   "PS/2"
```

and we replace it by

```
Option "Protocol"   "IMPS/2"
```

After 2 lines we add the following command also.

```
Option "ZAxisMapping"    "4 5"
```

We can press `Ctrl + X + C` in order to exit and we press `y` to save the changes.
If we did that through the X-Windows, we must restart our X-Server.

The editors that we can use in order to view or edit files in GNU/Linux are plenty. The most well-known are `jed`, `emacs`, `pico`, `nano`, `vi` and `vim` which can be used to modify files and `more` or `less`, which can be used only to view a file (read only mode).

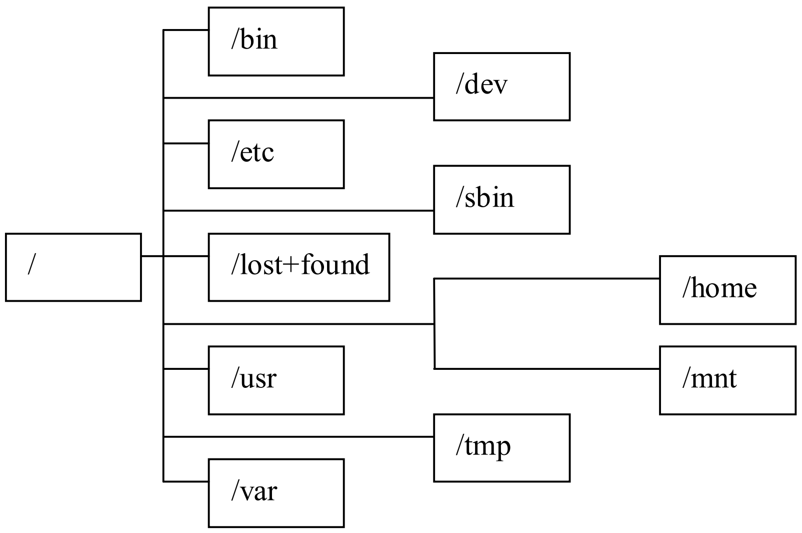
In the following table we will see some of the most important commands and files of GNU/Linux along with a small description in order to have a generic idea of what we are going to talk about in the following pages.
Before we go along we must describe another helpful functionality of GNU/Linux. Typing the name of a folder, file or command we can "auto complete" the rest of the word by clicking the tab button. If we double click the tab button will return all the possible commands and pathnames that can be used from that user at the specific location starting with the letter(s) we've typed.

File / Command	**Description**
jed /etc/passwd	Will open the passwords file into the jed editor
less /etc/shadow	Will open the shadow file into the less editor
ls | less	Will send the result of the ls command into the less editor

ls > /greg.txt	We can save to a file, called greg.txt in the top level directory, the result of the ls command
su	Usually in GNU/Linux we log in as simple user for security purposes. In order to change our login to super user (root) we must type su (without log off and log on again) and the system will prompt as for the root's password
su user	Change to "user" login
/usr/sbin/adduser	In order to add a new user to our system
/usr/bin/passwd	In order to change, modify passwords
pwd	This will return the current path that we are working on
ntsysv	Adjustments of what is going to start at boot time (RedHat)
ps -adl	This will display the processes currently running detailed
export PS1=	In order to change the prompt e.g. export PS1="Yes Master"
who	Lists all users currently logged in to the this machine
whoami	Display information for the current user login
id	Display id number and privileges of the current user login
find /-name <filename>	Searches the /file system for the requested <filename>. Wild characters such as * and ? can be used.
whereis <command>	Displays in which directories, included in the $PATH, the <command> is situated.
echo $PATH	Displays the path directories
/usr/bin/pine	A program that will allow as to access our e-mails
/usr/bin/elm	A program that will allow as to access our e-mails
/sbin/ifconfig	It's the same command as ipconfig for Windows ®
ifconfig eth0 up	Enable the first Ethernet card. The "ifconfig eth0 down" will disable it
/sbin/netconfig	Network configuration
/sbin/insmod module.o	Link a module with the Kernel
/sbin/rmmod module	Unload a module from the Kernel
/sbin/lsmod	We can see the modules that already have been loaded
/usr/sbin/cfdisk	Similar to fdisk partition table manager of Windows ®
less /etc/fstab	Which devices will be mount by themselves on GNU/Linux

	boot time. e.g root must be automount
less bin/dmesg	In that file we can see the outputs messages of the Kernel
whatis <command>	Displays a minimum help file for the command

In the diagram below we can see the directory structure in GNU/Linux. The top directory as we've already mention is called root. The following diagram along with a part from the description of the directories is from *(Introduction to UNIX security, Dr. Andrew Blyth, Lecture Notes 2002, MSc Information Systems and Network Security, University of Glamorgan)*



Starting from the top in the previous diagram we can see the **/bin** and **/sbin** directory which is the location for the binary files and/or script files. The **/dev** is the device directory containing special files that allow to the operating system to reference each piece of hardware. The **/etc** directory contains system configuration files and executables. A very important and useful directory is that which called **/lost+found**. Disk errors or incorrect system shutdown may cause files to become lost. Lost files refer to disk locations that are marked as in use in the data structures on the disk, but are not listed in any directory. The **/home** directory is a conventional location for user's home directories. The **`/mnt`** directory is a directory implemented in order for the users to mount some media, like our CD-ROM drive or a hard drive etc, into their file system. We will explain more about the `/mnt` directory in the following page. The following directory is the **`/usr`**, which, in the old days, was used to store personal configuration files for each user. Nowadays, it's initial usage has been replaced by the `/home` directory. It's current use is to store configuration files for most of the applications. The **`/tmp`** directory is available to all users and programs as a scratch directory. Last is the **`/var`** directory which we can think as the program files directory of Windows ®.

In order to access a device like the CD-Rom drive we must first "`mount`" the device. That means we must allocate a directory that will be a link to the filesystem of the specific device. Our devices are in the "`/dev`" folder. If for example, our first hard disk drive is called "`hda`", the second "`hdb`" etc. then in order to access the second partition of our 1st hard drive we must type "`hda2`".
Of course the floppy disk drive is "`fd0`" and if we have a second one it is called "`fd1`" (the devices are explained in the table at page 15).
The following command will connect the device "`fd0`" (1st floppy disk drive)
to the "floppy" directory inside the "`mnt`" directory.
"`mount /dev/fd0 /mnt/floppy`"

To disconnect the device we must type
"`umount /dev/fd0`"
or
"`umount /mnt/floppy`"

We could use "`mnt`" directory to mount a device. If we do that the "`mnt`" directory will be our device now until we `unmount` it. When we `unmount` the "`mnt`" directory we will be able again to see the contents of the original directory.

We can think of `mount` operation as a stack. We can `mount` several devices, one above the other, on a single directory. The last device we mounted (e.g. CD-ROM) is the device we are going to use from that directory. Unmounting the last device, automatically the directory is mounted to the previous device we had mounted (e.g. Floppy Drive). We will continue unmounting devices until we get the original directory "`mnt`". This is not useful very often, and thats why we use different directories for different devices. (MP (8) mount) [Reference a2, a3, a4, a5, a8]

Shell Scripting

The shell scripting in Linux is similar to vbscript of windows and is very easy to use in order to implement rapid scripts. At the Appendix B of the first sub-report (Page 81) there is a simple example that shows some useful commands in shell scripting. After writing our script we must change the privileges of the file in order to make it executable and be able to run it. A visual representation of the privileges of a file is showed below.

```
  r w x r w x r w x
 -|- - -|- - -|- - -|
```

r - Stands for read
w - Stands for write
x - Stands for execute

We can see the privileges of a file typing the command

```
ls -l
```

As we can see there are three triplets of `rwx`. The first triplet is the owner's privileges. The second triplet is the privileges of the owner's group and the third triplet is the other groups' triplets. Assigning a number for each letter e.g. r -> 4, w -> 2, x -> 1 we can create a unique number for each privileges we want to give. If we want read, write and execute for the owner we add 4, 2 and 1 which gives us 7. If we want read only access for the other groups' triplets we use the number 4. In our case we came up with the number 764.

```
  r w x r w x r w x
 -|- - -|- - -|- - -|
  |4 2  1|4 2 1|4 2 1|
   \     / \    / \    /
     7        6    4
```

We must type the following command to change privileges

```
chmod 764 file
```

in order to run the script we have just created we will type

```
./myFile
```

or we can move it into a directory that is included in the path and just type

```
myFile
```

[Reference a5]

Kernel:

Init runlevels

From the moment that an operating system starts booting until it reaches the run level we wish, it passes through the init faces. This faces are the init run levels which are 6 from init 0 to init 6. Below we can see what is happening to the operating system as it passes from the one stage (run level) to the other.

init 0 - (init 0 and init 6 are the same thing)

init 1 - single user (not multi-user), (no network)

init 2 - always empty (in order to develop a custom run level)

init 3 - multi-user (without x) plus network

init 4 - Depends of the distribution (maybe either empty or represent multi-user, graphical environment with networking)

init 5 - Depends of the distribution (maybe either empty or represent multi-user, graphical environment with networking) init 6 - Reboots the machine.

Recompile your kernel

In order to use some hardware devices that we might have, or to be able to “see” the ntfs file system of Windows ® from GNU/Linux we must recompile our kernel. Actually that recompile will add into the kernel or link to the kernel modules that are necessary in order to be able to perform the above operations. These modules are either drivers for the hardware devices or pieces of kernel code in purpose to do a specific task.

In the following pages it will be explained as better as we can how to recompile the kernel.

First of all, we have to make sure that linux’s source code is installed into our system. If we are running an rpm based distribution (such as Red Hat) and the kernel source package is `linux-source.rpm`, we can type

```
rpm –v linux-source.rpm
```

Another way, used on every distribution, to check if we have the kernel source installed, would be to change into the kernel source directory using

```
cd /usr/src/linux
```

and check if the source code is there.

As soon as we verify that we have the source code, we can do a

```
make clean
```

to remove any temporary and unwanted files from that directory. Then we can run one of the three *.config*[3] editors by typing one of the following command lines:

```
make config
```

OR

```
make menuconfig
```

OR

```
make xconfig (only for X)
```

The *.config* file is a configuration file, which we can see by typing

```
less ./config
```

lining in the `/usr/src/linux` directory and it stores every information the "`make`" program needs to know about which kernel modules to compile with the kernel and which to compile as loadable modules. After editing this file, we can save the new kernel configuration and check if there any modules included in the kernel that are depended from some others which were not included with

```
make dep
```

If the previous command is successful we can move into compiling a compressed image of our new kernel by typing

```
make bzImage
```

and then compiling and installing the modules with

```
make modules && make modules_install
```

[3] files starting with a dot [.] are hidden and can be seen with the ls –l command

Finally, if everything has gone right during the compiling and installing process, we can add the new kernel image "`bzImage`", situated into `/usr/src/linuc/arc/i386/boot/`, into our boot loader (such as `lilo` or `grub`) and then reboot into it. [Reference a4, a5]

The LILO boot loader

One of the most widespread boot loaders, for Linux partitions, is LILO. Lilo reads the configuration file `/etc/lilo.conf` and then stores the boot parameters in the first 512 bytes of a partition or in a hard drive's MBR (Master Boot Record). When a PC boots, it reads off those 512 bytes and then loads the kernel image described there. In order to change the configuration file we can either run

```
liloconfig
```

or manually edit it by typing the following command

```
jed /etc/lilo.conf
```

In order to add another kernel image option for example `/bzImage` we have to declare some variables (the author places them at the end of file). First of all, the image's path. This can be done by typing into the `lilo.conf`

```
image= /bzImage
```

Then the root partition should be stated with

```
 root= /dev/hda3
```

Additionally, we must assign a name to the new kernel image, in order to recognize it at boot time in the `lilo` prompt.

```
 label= Slackware.new
```

Finally, because Linux at boot time runs some diagnostic tools on the root partition, in order to prevent any damage to the file system, we initially mount it as read only. So, we add the following command.

```
 read-only
```

If we want `lilo` to be installed on the MBR the following line should exist on the top of the file.

```
#...
boot= /dev/hda
#..
```

Moreover, in order to have a time period equal to 50 ms in order for the user to be able to choose which kernel to boot we must add the following lines in the `lilo.conf` file.

```
prompt
delay = 50
```

As a final point, we have to type the command

```
lilo
```

in the command line.

[Reference a2, a4]

Kernel programming

As it is already mentioned in the introduction, most of the kernel source code is written in C programming language. The best approach, by the author's point of view, to develop a Linux kernel module is by using C.

When programming for the kernel, the standards of ANSI[4] C still exist. However, there are some differences between kernel space and user space programming. There are some libraries used only to be linked with kernel modules and consequently there are some functions that will only be recognized by the kernel.

First of all, we have to go over the structure of standard kernel module. Each module should have one "init" function and one "exit" function. Each time the module is loaded the kernel executes the "init" function, which usually checks for the availability of some resources and/or registers some of them. When the module is unloaded the "exit" (cleanup) function is executed in order to un-register everything that was previously registered by the module and leave the kernel as it was before. If the "init" function fails (returns a minus integer number) the module is considered not to be loaded. On the other hand, a cleanup function should never fail, that's why the programmer should make sure that every procedure included in this function must be successful.

As everything in the GNU/Linux operating system is treated as a file by the kernel, then a file operations structure should exist in order to declare which operations are permitted and how the kernel should react when each of them is called by the user space. Whenever file operation is performed on a file by the application layer, the kernel searches for the module that has been register with this specific file and responds as its file operations structure describes.

[4] American National Standard Institute

After the C program has been developed, it must be compiled (but not linked) using a ANSI C compiler. Using the gcc compiler this can be done with the following command line

```
gcc  -c module-name.c
```

This will produce our ELF binary *module-name.o*. This should be linked then into the kernel by typing

```
insmod module-name.o
```

Now the module should be loaded into the kernel. We can verify that by listing all the modules loaded into the kernel with

```
lsmod
```

and checking if it is in the listing.
If someone would like to unload the module, he should type

```
rmmod module-name
```

[Reference a1, a3, a5, a6, a9, a10]

ISA Input Output Card

A BUS, in the computer world, is defined as the medium around which the data from a device to another one circulate or of a device to the memory and/or the CPU. A card can communicate with the CPU using a rank of ports, possible interrupts and DMA(s)[5]. In the following points, it is possible to be seen as they are assigned or not in each type of bus architecture.

[5] Direct Memory Access

The first bus architecture that was implemented in PC architecture was ISA. At first it was an 8-bit medium but it quickly evolved in 32-bits. This indicates that every time, in each clock cycle, the bus was able to transfer one and two bytes respectively. Even thought technology enables to increase the bus frequency for ISA, backward compatibly issues force it to be stable at 8.33 MHz. A 16-bit ISA card is able to support a maximum of 16 megabytes of transference per second for all the devices simultaneously, a record which has been overcome long time ago.

The PCI architecture is the newest and most widespread, now days, bus architecture. Norm PCI, indicates that a 4 BUS PCI is solely of 4 slots, but allows us to connect another 4 slots in groups, by means of "PCI bridging". Also, PCI is a 32-bit bus with a frequency of 33 MHz. This means that the maximum transfer rate can be 133 megabytes per second. This is more than double the maximum speed of an ISA bus. Additionally, IRQ[6] sharing can be supported by this architecture, something that PnP[7] and Windows® take a great advantage of.

The author initially had chosen to use a PCI card for the purposes of this project. Unfortunately, a posting error forced the use of an ISA Input Output card, whose specifications will follow.

The exact model code of the card is PCL-731, which is a 48-bit Digital Input Output (DIO) card. This means that it is equipped with 48 pins, which can be used as inputs or outputs to and from the card. Several configurations can be made in order to set some pins as inputs and some others as outputs. In addition, a "8250 family" microcontroller is attached to the card and is responsible for the algorithm of the digital signal processing of the data coming into and going out of the card. This microcontroller is fully programmable and can change completely the functionality of the card. Still, the two IRQ's , of the card ,as well the trigger level of the them can be manually selected with the use of two jumpers.

[6] **I**nterrupt **Req**uest Line
[7] Plug and Play

Furthermore, the card has two 50-pin OPTO-22 compatible connectors, where the 48 input/outputs are situated, divided into six 8-bit ports I/O ports, along with some registers. A 50-way IDC connector may be easily connected to them and then transfer the data into an IDC cable connected to an external device. Also, the card is equipped with 48 LED's[8], corresponding to the 48 I/O pins, showing the high or low logic for each pin. The logic High voltage for each input varies between 2.0 and 5.25V and the logic Low voltage for the previous is above 0.0 and below 0.8V. The logic High output signal is 2.4V and the logic Low of the same output is 0.4V.

As it was mentioned before, the 48 I/O pins are divided into six 1-byte ports. Each one of them is assigned to a port address always in respect to the base address of the card which can be in the range from 0x0 to 0x3F8. Below a table of all the I/O ports and their corresponding address is shown.

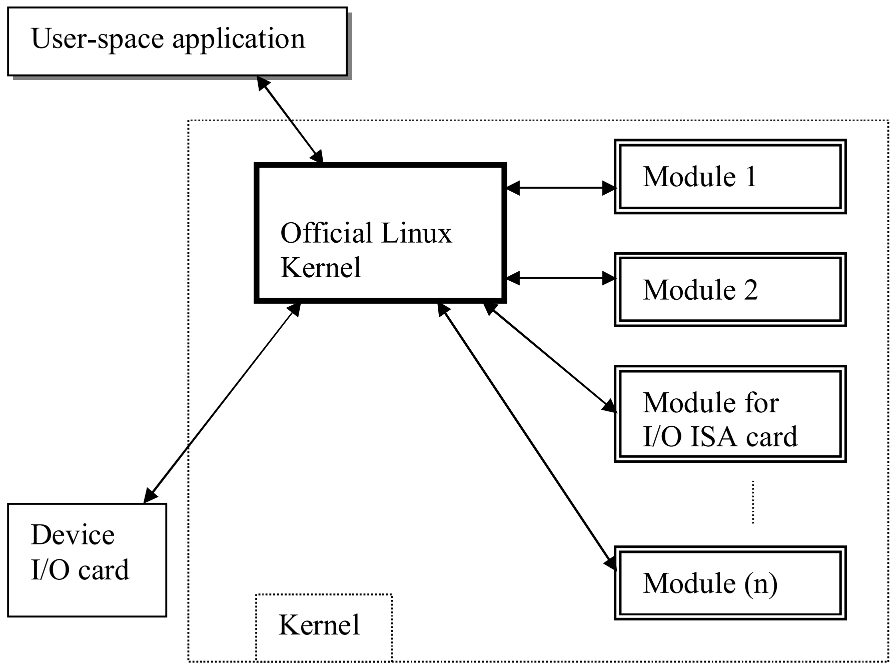
Address	**Port**
Base + 0	**AO**
Base + 1	**B0**
Base + 2	**C0**
Base + 3	**CFG REG**
Base + 4	**A1**
Base + 5	**B1**
Base + 6	**C1**
Base + 7	**CFG REG**

[Reference a5, a11, a12, a13, a14]

[8] Light Emitting Diodes

How a module works:

In the following diagram we can see how a module is linked to the kernel and how it manipulates requests from the user-space program or the device through the kernel in order to perform the requested operations.



Once we have developed a module we must link it to the kernel. Using the `insmod` command and the name of the ELF[9] file we can load our module to the Kernel (e.g. `insmod isa731.o`). Typing the `lsmod` command we can see if the module was successfully loaded or we can use the `dmesg` command to see what messages the module sends to the kernel. When a module is linked to the kernel, it registers a range of ports and a major number. This range of ports is going to be accessed only by our module and that's where the device is connected. The major number is a unique number in order for the kernel to identify the specific device. The module doesn't know the device's representative filename in the file system, but just its major number. Any device with the same major number is manipulated by the same module.

9 (Executable and Linking Format)

Then, they are distinguished by a second numerical prefix, which is the minor number. The user-space program, assuming that it is trying to perform an operation using the I/O hardware device, sends to the kernel a request in order to open the specific device. The kernel is looking for the specific module that is loaded for that specific device, matching the major number from the device and the module. The module holds information about what the kernel should do when a request reaches it. When the request arrives to the kernel it searches for the appropriate module to manipulate the request. The module "tells" to the kernel what to do with the particular request and then the kernel "tells" to the device to do it. Finally, the kernel will return to the user-space program any reply that comes from the device, if any. The same operation takes place when a request comes from the device in the opposite direction. [Reference a1, a3, a5]

Devices

Linux has a way to identify the type of devices that correspond to the system. These devices, which normally exist into the `/dev` directory, can be accessed by modules that consist of three different classes. These three classes of modules are the char modules, block modules and network modules. Every module that is included into the kernel or a module that we might want to load manually must have this type of information. A small description is provided in the following lines in order to understand how the system recognizes each of the three classes of modules.

A character device can be accessed as a stream of bytes. This behavior is implemented by a char device. This driver usually implements four types of system calls, the open, close, read and write system calls. Char devices are accesses by means of file system nodes[10]. A char device differs from a regular file just in the fact that in the regular file you are always able to move back and forth, while char devices are just data channels. This means that you can only access sequentially.

[10] E.g. /dev/tty1 and /dev/lp0

A block device, as well as char devices are accessed by file system nodes in the /dev directory. A block device is able to host a file system, for example a disk. Linux permits the application to read and write a block device like a char device, and allows the transfer of any number of bytes at time. Consequently, block and char devices are different just in the way the kernel manages internally the data and therefore in the kernel/driver software interface. The differences between them are noticeable to the user. A block driver gives the kernel the same interface with a char driver. Furthermore, it offers an additional block-oriented interface which is invisible to the user. In addition it can offer applications opening the /dev entry points. However this interface is crucial to have the ability to mount a file system.

Every network transaction is made through an interface. This means that there is a device that can exchange data with other hosts. In most cases an interface is a hardware device; however it is possible to be a pure software device[11]. The role of the network interface is to send and receive data packers, driven by the network subsystem of the kernel, with no knowledge of how individual transactions map to the actual packets being transmitted. A network interface is not easily mapped to a node in the file system due to the fact that it is not a stream-oriented device. The Linux way to supply access to interfaces is also by assigning a unique name to them[12]. Nevertheless that name does not have a corresponding entry in the file system. The kernel and a network device driver have a completely different communication from char and block drivers. The kernel calls functions are associated to packet transmission as a replacement for reading and writing.
[Reference a3, a4]

Device Major Number

In order for a device to be recognized by the kernel among the other devices, each device is assigned a unique number. This number is called the "device's major number". This can be set during the init function of the module. The programmer has the choice to either allocate it dynamically, that is let the kernel decide about the

[11] e.g. loopback interface
[12] e.g. eth0

major number (kernel 2.4.x), or manually, by specifying the number in the source code of the module or at load time. In the current design of the module the second approach was used. This is because, in order to make a node for the card in the file system, we must know the major number of the device assign during the load of the module. In case we had used the third option, (let the kernel decide the major number), we should look into the /proc/devices file in order to find the major number. Instead, the manual allocation of the major number was used and the only thing we should be aware of is that this number is not in the `/usr/src/linux/Documentation/devices.txt` and is not already registered to the kernel. The chosen number to be used as major number for the ISA card is 42.
[Reference a3, a5]

Device name

A unique name should be invented in order to associate it with the device. This would be the modules name as appeared at the output of the command `lsmod` and the node's name appeared under `/dev/`. The author of this project chooses the name **`isa731`** as a device name.
[Reference a3, a5]

Usage Count

Usage count is a variable that we change each time we using the device. This is for the reason that we don't want no other module to access the device when we are using it. Using the functions MOD_INC_USE_COUNT and MOD_DEC_USE_COUNT we can increase and decrease the usage count. This operation takes place when we use the file operations open and release. When we open the device we must increase the usage count and when we release the device we must decrease the usage count.
[Reference b3, b5]

How can we create a node

As we already mention a node is a file that represents a device into the `/dev` directory. In order to create a node we must use the following command before load our module to the kernel.

```
mknod DEVICE_NAME c MAJOR_NUM MINOR_NUM
E.g. mknod /dev/isa731 c 42 0
```

That command creates a node into the /dev directory named `isa731` that is a character device (`c`) with `major` number 42 and whose `minor` number is 0.

The minor number can help to open the same device in a different way. Furthermore, if we have on our system 2 identical devices e.g. two ISA I/O cards, we can access them using the same module but the node for each card will have a different minor number.
[Reference a3]

Loading modules automatically

An especially useful feature is the kernel daemon “`kerneld`”. The kernel can load needed device drivers and other modules automatically without manual intervention from the system administrator. If the modules are not needed after a period of time (60 seconds), they are automatically unloaded as well. In order to take use of the `kerneld` we must turn it on during the kernel configuration along with the System V IPC option. The kernel configuration is the kernel recompile that we discussed in the first sub-repost of this project.
[Reference b3, b4]

User Space Program:

In this section of the report we would like step through and explain some basic points of the user’s space source code.

The source code of the user space is included in the appendix 2 (Page 50).

The user space program, in order to send or receive data from the device, must use the module as the middle man. The module in order to access the device uses the special function to write to ports. From the user space program we must use normal file operations on the devices representative file. In order to use the file operations through the module that we developed we must open the file. This, will open the device in order to be used (the `open_card` function of the module will be executed) or if the device does not exist it will print a message.

Then, we have a loop in order for the program to prompt us continuously if we would like to read from or write to the device. Pressing the Ctrl + C the program will exit. In each case (read or write) we call a function. These functions are `read__` and `write__`. In the `write__` function using the file descriptor, acquired from the open function, we write a value to the file. At that time the `write_card` function of the module is executed. The `read__` function reads from the file the input value from the device and causes the `read_card` function of the module to execute. When the program is terminated by the software interrupt `Ctrl + C`, the kernel closes all the files with file descriptors used by that program and the module executes its `close_card` function.
[Reference a3, a5, a6]

How the ISA I/O card was programmed:

The ISA I/O card PCL-731 has two 50 pin IDC connectors. In the following picture we can see the IDC connectors. According to the user's manual of the ISA card we must register two addresses in order to use them for writing and reading from the device. The student, after some testing, decides to use the CN1 IDC connector to write to the device and the CN2 IDC connector to read from the device.

The pins 49 are the pins that provide the 5Volts. All the GND are the ground pins and the rest are the pins that we can send or read bits (logical 1 or 0).

CN1

PC07	1	2	GND
PC06	3	4	GND
PC05	5	6	GND
PC04	7	8	GND
PC03	9	10	GND
PC02	11	12	GND
PC01	13	14	GND
PC00	15	16	GND
PB07	17	18	GND
PB06	19	20	GND
PB05	21	22	GND
PB04	23	24	GND
PB03	25	26	GND
PB02	27	28	GND
PB01	29	30	GND
PB00	31	32	GND
PA07	33	34	GND
PA06	35	36	GND
PA05	37	38	GND
PA04	39	40	GND
PA03	41	42	GND
PA02	43	44	GND
PA01	45	46	GND
PA00	47	48	GND
+5 V	49	50	GND

CN2

PC17	1	2	GND
PC16	3	4	GND
PC15	5	6	GND
PC14	7	8	GND
PC13	9	10	GND
PC12	11	12	GND
PC11	13	14	GND
PC10	15	16	GND
PB17	17	18	GND
PB16	19	20	GND
PB15	21	22	GND
PB14	23	24	GND
PB13	25	26	GND
PB12	27	28	GND
PB11	29	30	GND
PB10	31	32	GND
PA17	33	34	GND
PA16	35	36	GND
PA15	37	38	GND
PA14	39	40	GND
PA13	41	42	GND
PA12	43	44	GND
PA11	45	46	GND
PA10	47	48	GND
+5 V	49	50	GND

By default the ISA I/O card is set for a base address of 0x300. The author of this project has changed the base address of the card and was set to be 0x200. In order to set the CN1 IDC connector that will be used for writing to the device we send to the address 0x203 (base address + 3) the binary number 10000000.

When we write a byte to the device we are setting the pins PA00 – PA07 to logical 1 or 0, depending to the byte we want to write. For example, if we write to the card the binary number 11110001 the pins PA07, PA06, PA05, PA04, and PA00 will be set to logical 1. The pins PA03, PA02, PA01, will be set to logical 0. Sending to the base address 0x200 a binary number after we set which port range will be used for output it will be written to the pins PA00 – PA07.

At this point we must mention that when we write a value to the device that value stays to the device until we overwrite it with a new value or if we reset the card (reboot the computer).

In order to read from the device we use the CN2 IDC connector and we must set the port range that we will use for that operation. The base address 0x207 (base address + 7) will be used for read (input) from the device. Sending the binary value 10010000 to the base address 0x207 will be set to read.

When we read from the device we are reading the value that exist on the pins PA10 – PA17. Reading the 0x204 address (base address + 4) will return the input from the device. [Reference a7, a8]

Step through the source code of the Module:

In this section of the report we will step through and explain some basic points of the module's source code.

The source code of the module is included in the appendix 3 (Page 53).

In order for the user-space to be able to access the ISA I/O card which is a character device, it must use the file operations. The file operations used in this module are Open, Release, Write and Read.

At the top of the module's source code we can see the libraries used and defining some constant variables. Defining the module as `__NO_VERSION__` means that the module has no problem to cooperate with any kernel version. Then, we can see the functions that were used and the declaration of the init and exit functions.

The init function is the first function that will run when the module is loaded into the kernel, correspondently, the exit function is the function that will run when we unload the module from the kernel. These functions "`init`, `exit`" will register and un-register the device using the corresponding functions `register_device` and `unregister_device`.

The "`set operation mode`" function will write a specific byte to specific ports of the card in order to set the input and output ports of the card. Those ports are called the registers.

Finally, the most important part of the module is the actions that are being performed by the file operations. In the first case we have the `open_card` file operation which checks if the device is in use, in order to print a message to the kernel if it is, and then increases the usage count. If the function was executed successfully it will send a second message to the kernel. In order to see these messages we must type the `dmesg` command which provide us with all the kernel messages.

For the release file operation which in our case we named `exit_card`, the only thing we must do is to decrease the usage count. Failure to do that will have as result to set us unable to unload the module from the kernel. In order to unload the module we would have to restart the computer.

The write function, using the base address that was set for output will send to the device a value. That value is the incoming value from the user space program. In order for the module to read the value that is entered in the user space program we save that value to a variable. The module will read this value using the pointer of the value and finally output it to the device.

The read function, is trying to get the data from the device reading the specific base address that is set for read. It then passes it to a pointer which is returned to the user space.
[Reference a1, a3, a5, a6]

Writing a script for easier manipulation:

During the research for the first sub-report the author learned how to write simple script files for Linux. That came to be a very helpful when the time of compiling the modules the student wrote a script file for easier manipulation.

```
#!/bin/bash
gcc -D __KERNEL__ -O2  -Wall -I"/usr/src/linux/include/" -c labcard-alt.c -o labcard-alt.o
```

```
#!/bin/bash
gcc -Wall user.c -o user.exe
```

If we copy the script files into the `/sbin` directory we can run them without typing the path. That means that will make the operation system to execute the script files as another system command. (e.g. `ifconfig`)
[Reference a5]

Interrupts:

Interrupts are system calls that are being send to the CPU to let the processor know when something has happened. The hardware (a device) sends a signal to the processor in order to requests for the processor's attention. Generating and manipulating interrupts is a very complex procedure. For that reason interrupts weren't used for the development of the module but this small paragraph was included in order to explain in general how interrupts work.

When a hardware interrupt is generated by a device the CPU sends an IRQ signal to the operating system's kernel. The kernel then according to it's configuration can ignore the interrupt and continue it's normal schedule of operations (e.g. Round Robin[13]) or check it's IRQ table to see if any of the kernel modules has been registered with that interrupt value. If that is the case, it will then probe the module in order to inform it about the situation. It's up to the module then to take further action if needed.

The ISA I/O card provides some jumpers that will allow us to set the interrupts. The first two jumpers (JP5, JP6) will help us to register what kind of IRQ levels the device will use. Jumpers JP1 and JP2 will identify the type of the interrupt triggering. There are two types of interrupt triggering, the rising edge and the falling edge. In the rising edge the interrupt of the device is set to send the logical 0 and when an interrupt occurs it rises to the logical 1. In the falling edge when an interrupt occurs the signal falls from the logical 1 to the logical 0. Using the jumpers JP3, JP4 we can enable, disable or use the S/W setting of the interrupts. The S/W option is there to allow us to control the interrupts by other external interrupts.
[Reference a1, a5, a11]

[13] http://choices.cs.uiuc.edu/~f-kon/RoundRobin/node1.html

The overall progress summary

The student had no previous experience in programming with Linux. Actually the first time that he used Linux was at the second year of his studies in one of his modules about Operating Systems. The main idea of the project seemed exciting and challenging. The only knowledge the student had around the subject when the project started was the C programming language used in the Windows® environment. The student did his first installation on Linux, partially completed, during the summer vacation before he started the third year of his degree.

At the beginning of the project the student had to install Linux (successfully this time) and research how it works, learn the basic commands and understand the structure of the operating system. The man files and the apropos database were one of the most important tools in order to get started with Linux.

The development started by writing small pieces of code in order to see how the C++ programming language works with Linux. The `gcc` program, as long as a great amount of shared libraries from `/lib/`, where used for that purpose. In order to develop the source code for these small applications the jed editor (in `emacs` mode) was used. After familiarizing with user space programming in GNU/Linux, kernel programming was introduced. The student did a lot of research on the matter before attempting to actually write any pieces of code.

The next step of the research was to find out what the structure of a device driver is and how a device driver works. After accomplishing that goal and being able to identify the basic parts of a device driver and understand their purpose, the model of a very simple kernel module was designed. The module was supposed to send a simple message from the kernel to the user space each time it was loaded and/or unloaded. After spending some time on that, the module was finally doing what it should be when "`insmoded`" to the kernel. After altering the code of the module and observing the results (kernel messages, kernel panics, kernel oops) for a short period of time, the kernel-module architecture was finally well understood.

Consequently, the file system's structure was studied in order to better understand the way a user space program access a resource and the means that the kernel uses to represent them to the file system. Concepts like major and minor number were introduced and the whole picture was getting clearer. After being able to create a node into the `/dev` directory and link/unlink a module to the kernel, according to the research undertaken [Reference a3, a5 (chapter 21)] the next stage was to add pieces of code to the basic structure of the file operations. Several mistakes occurred during the implementation and they were solved using the examples of the books that were used as prototypes. During the implementation of the project several pieces of source code were shown to the supervisor in order to be sure that the project was on the right path to accomplish its target. Fortunately, the module was implemented successfully.

Testing of the project

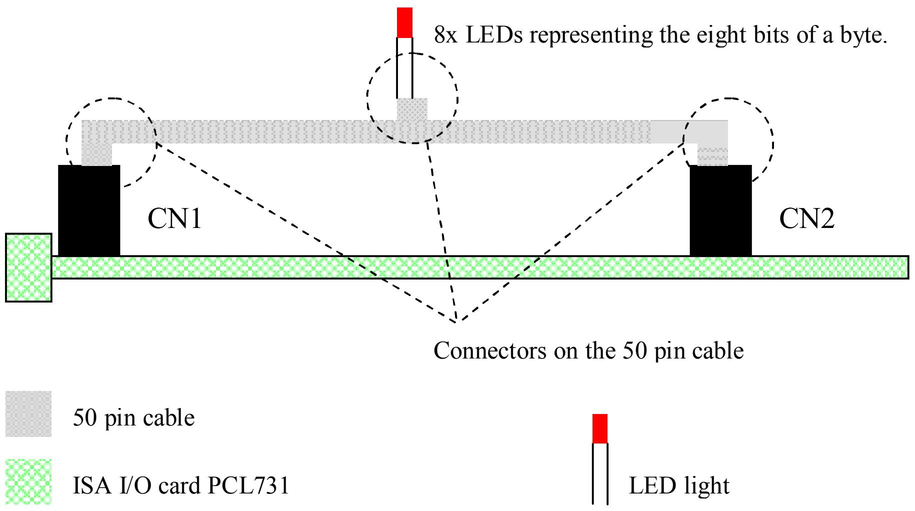
During the implementation of the module several tests took place. The read function was the first that was tried to be implemented. After consulting the supervisor of the project, it was decided that the read function should be implemented first. Several tries later, the function was returning the value 0xFF (255) which were assumed is the default value coming from the card.

The next step was to send a default value to the card from the module without using the user space program. When that goal was accomplished, pieces of code started to be added into the user space program in order to make it communicate efficient with the module. Working in parallel tasks trying to develop the read and the write function, problems occurred between the transaction of the variables between the module and the user space program. A variable was successfully been written to the device using the module but not through the user space.

After several tests the user space program was sending a number to be written to the device successfully. In order to check that the module received the correct variable an additional `printk` of that variable was added into the code. Conceptually if the module received the correct value someone could see it using the `dmesg` where all the messages from the kernel are displayed.

The next step was to try to read that value from the module and then from the user space program. Unfortunately, even after the correction of some programming mistakes the incoming value was still 0xFF and it was also the value that could be seen from the user space application.

After contacting the supervisor for any piece of advice he could provide on the problem occurred some extended testing took place. A fifty pin cable and led lights were bought in order to have a real representation of the values that exist on the card. The cable was connected from the CN1 connector of the card to the CN2. The cable had a third connector that was used for connecting the led lights. Actually it was like sniffing the signal passing through the two connectors in order to have a real representation. (See figure below)


8x LEDs representing the eight bits of a byte.
CN1
CN2
Connectors on the 50 pin cable
50 pin cable
ISA I/O card PCL731
LED light

(*See Appendix 5 at Page 59 for a real picture*)

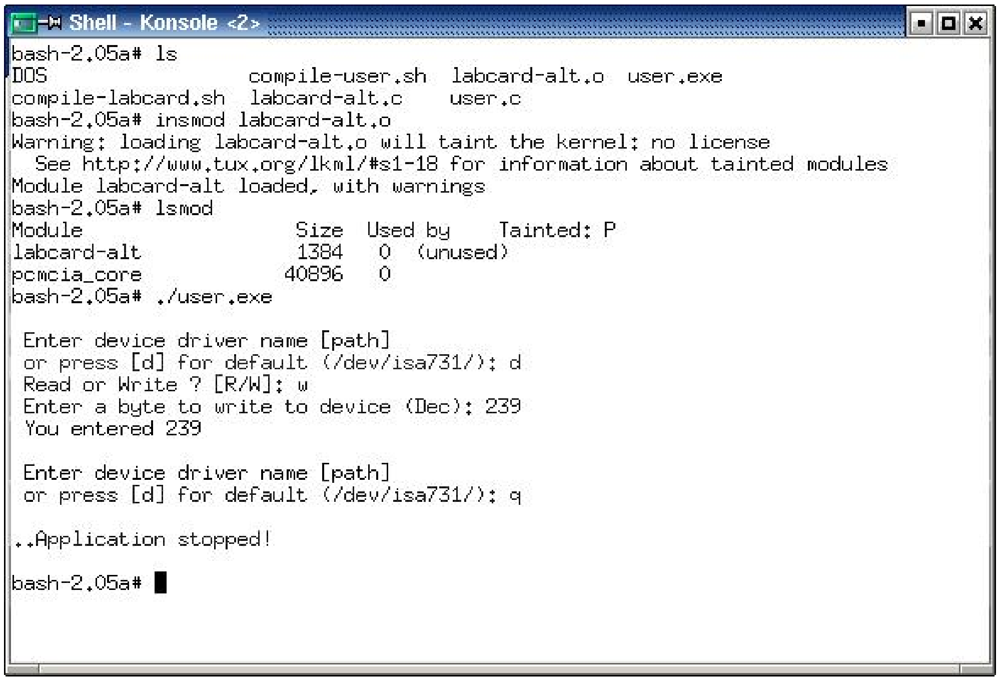
The led lights proved that the correct values were sending to the card, but the read function wasn't working properly. Several tests occurred in order for this mistake to be fixed and the advice of the supervisor was asked. The author also contact the manufacturer company of the ISA card for any special tips they might have for that problem but no answer was received.

The User's manual of the ISA card had the source code of a very simple application written in C that was writing/reading values from/to the card in order to test it. This program needed an MS-DOS partition in order to be used. The author, using the Borland turbo C compiler and the source code from the user's manual, created the executable file of that program. Also, an MS-DOS 6.22 boot disk was created including the executable file in order to test the card. Finally, the output of that executable file was identical to the one from the Linux user – space program.

The supervisor was informed for that result and after a couple of meetings with the student come up with the result that there should be a hardware problem with the ISA card and that this is the reason for receiving the same results from both applications (Linux Kernel module, MS-DOS executable file).Consequently, it was agreed both from the supervisor's and the developers point of view that the function "`read`" of the microcontroller on the card was faulty and was outputting always the same number.

Screenshots – System goes live

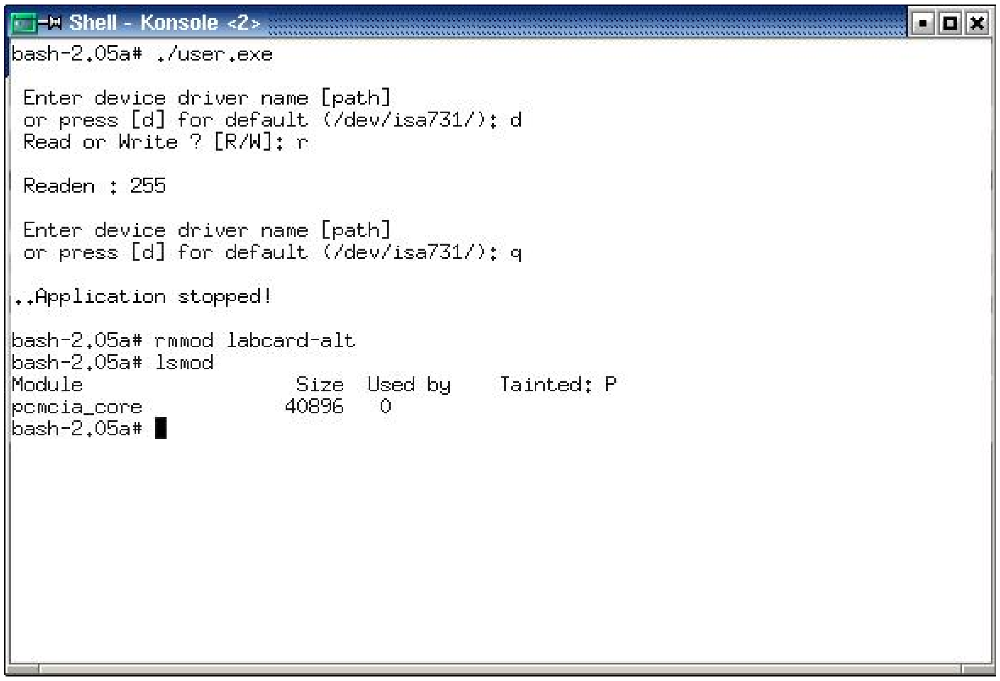
The following screenshots showing the loaded module and the user space application running.



```
bash-2.05a# ls
DOS                 compile-user.sh  labcard-alt.o  user.exe
compile-labcard.sh  labcard-alt.c    user.c
bash-2.05a# insmod labcard-alt.o
Warning: loading labcard-alt.o will taint the kernel: no license
  See http://www.tux.org/lkml/#s1-18 for information about tainted modules
Module labcard-alt loaded, with warnings
bash-2.05a# lsmod
Module                  Size  Used by    Tainted: P
labcard-alt             1384   0  (unused)
pcmcia_core            40896   0
bash-2.05a# ./user.exe

 Enter device driver name [path]
 or press [d] for default (/dev/isa731/): d
 Read or Write ? [R/W]: w
 Enter a byte to write to device (Dec): 239
 You entered 239

 Enter device driver name [path]
 or press [d] for default (/dev/isa731/): q

..Application stopped!

bash-2.05a# 
```

First, the module is linked to the kernel using the *insmod* command followed by the name of the compiled binary file (*insmod labcard-alt.o*). The *lsmod* command shows the loaded modules. The next step is to run the user space application which will prompt us for the device name. Pressing the “d” key for the default path (`/dev/isa731`) the question if we would like to Read of Write from/to the ISA card will come up. By pressing w which stands for “Write to the card” the program will prompt to enter a Decimal number to be written to the card. The program loops continually in order to perform new operations (read / write) but by pressing the `q` letter it exits. (real pictures of the ISA I/O card connected on the computer can be found in Appendix 5)

The following picture is showing how a value can be read from the device. Unfortunately, the ISA card sends always the default value 255.



```
bash-2.05a# ./user.exe

 Enter device driver name [path]
 or press [d] for default (/dev/isa731/): d
 Read or Write ? [R/W]: r

 Readen : 255

 Enter device driver name [path]
 or press [d] for default (/dev/isa731/): q

..Application stopped!

bash-2.05a# rmmod labcard-alt
bash-2.05a# lsmod
Module                  Size  Used by    Tainted: P
pcmcia_core            40896   0
bash-2.05a# 
```

The last step is to remove the linked module from the kernel using the *rmmod* command (*rmmod labcard-alt*). Once again, using the *lsmod* command we can see the loaded modules and verify that the module was successfully removed.

Conclusions

In conclusion the project can be considered successful as the primary target of it. That is building a device driver for the ISA I/O card and a user space program, have been met in a satisfactory level. There are things that could be improved, such the quality of the source code like being more optimized, add the module in the kernel source tree and make it visible by the *menuconfig* and be able to use interrupts.

Considering that from the beginning the nature of the project was very challenging the developer accomplished to rich the best level of development he could.

References:

[a1]Daniel P. Bovet, Marco Cesati 2000. *Understanding the Linux Kernel. O'Reilly.*

[a2]Bill Ball 1998. *Teach Yourself Linux in 24 Hours: Second Edition. Sams Publishing.*

[a3]Alessandro Rubini & Jonathan Corbet 2001. *Linux Device Drivers: Second Edition. O'Reilly.*

[a4]Matt Welsh, Matthias Kalle Dalheimer, Lar Kaufman 1999. *Running Linux. O'Reilly.*

[a5]Richard Stones & Neil Matthew 2001. *Beginning Linux Programming.Wrox.*

[a6]Jesse Liberty, David Hovarth 2000. *Teach Yourself C++ for Linux in 21 Days. Sams.*

[a7]Don Anderson, Tom Shanley 1995. *ISA System Architecture. Addison Wesley*

[a8]*Introduction to UNIX security*, Dr. Andrew Blyth, Lecture Notes 2002, MSc Information Systems and Network Security, University of Glamorgan.

[a9]Linux Journal, October 2001, issue 90, How to write a Linux USB device driver p.24.

[a10]Linux Magazine, September 2002, issue 23, language of C p.66.

[a11]Advantech. User's Manual for the ISA card PCL-731. 48-bit Digital I/O card.

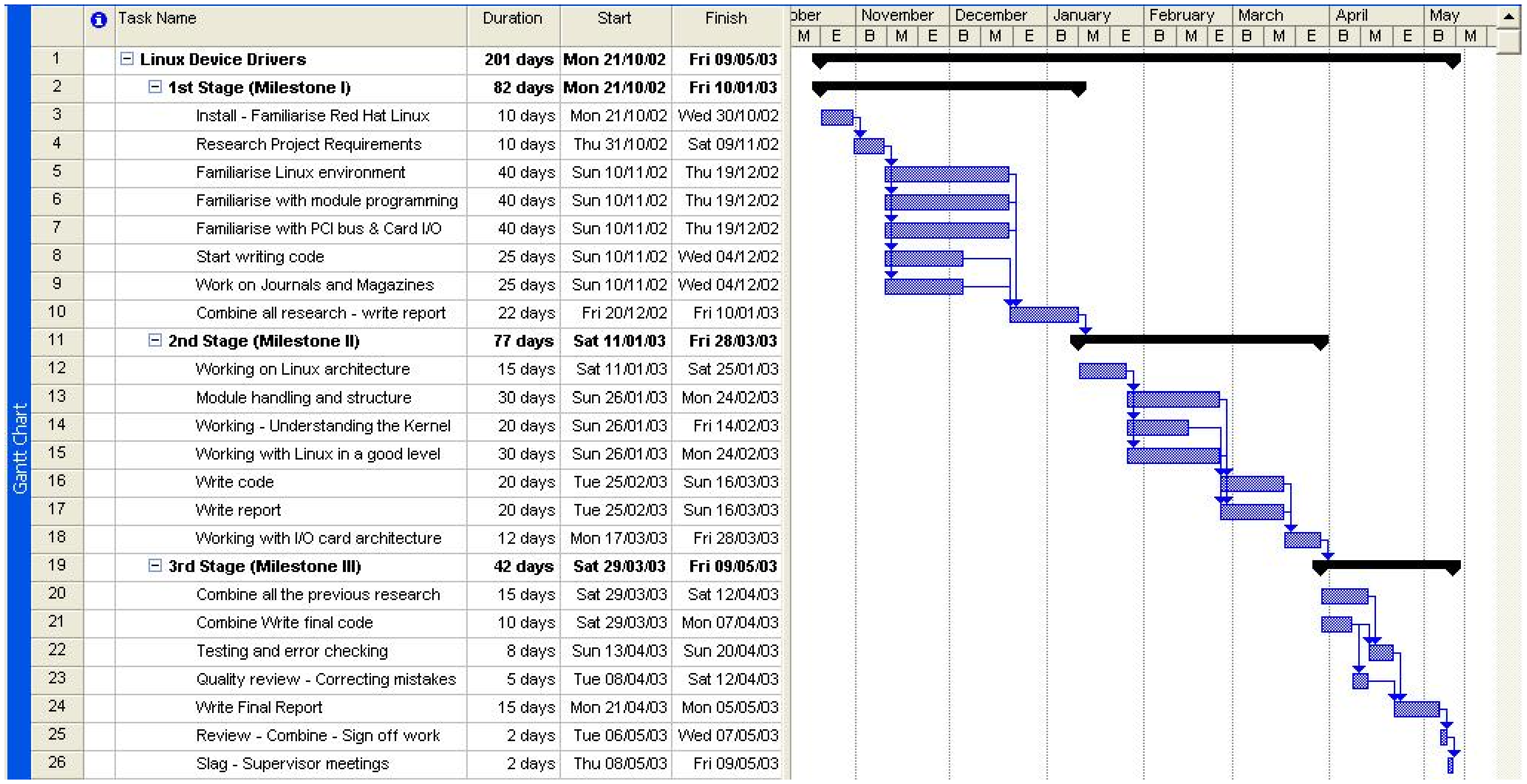
[a12]http://www.cse.ogi.edu/class/cse521/2002fall/Lec02-ISA.ppt

[a13]http://sunsite.tut.fi/hwb/co_ISA_Tech.html

[a14]http://www.cs.arizona.edu/computer.help/policy/DIGITAL_unix/AA-Q0R6C-TET1_html/TITLE.html

All the web links used were last accessed on Monday 12/May/2003

Appendix 1 – Project Time Plan



		Task Name	Duration	Start	Finish
1		⊟ **Linux Device Drivers**	**201 days**	**Mon 21/10/02**	**Fri 09/05/03**
2		⊟ **1st Stage (Milestone I)**	**82 days**	**Mon 21/10/02**	**Fri 10/01/03**
3		Install - Familiarise Red Hat Linux	10 days	Mon 21/10/02	Wed 30/10/02
4		Research Project Requirements	10 days	Thu 31/10/02	Sat 09/11/02
5		Familiarise Linux environment	40 days	Sun 10/11/02	Thu 19/12/02
6		Familiarise with module programming	40 days	Sun 10/11/02	Thu 19/12/02
7		Familiarise with PCI bus & Card I/O	40 days	Sun 10/11/02	Thu 19/12/02
8		Start writing code	25 days	Sun 10/11/02	Wed 04/12/02
9		Work on Journals and Magazines	25 days	Sun 10/11/02	Wed 04/12/02
10		Combine all research - write report	22 days	Fri 20/12/02	Fri 10/01/03
11		⊟ **2nd Stage (Milestone II)**	**77 days**	**Sat 11/01/03**	**Fri 28/03/03**
12		Working on Linux architecture	15 days	Sat 11/01/03	Sat 25/01/03
13		Module handling and structure	30 days	Sun 26/01/03	Mon 24/02/03
14		Working - Understanding the Kernel	20 days	Sun 26/01/03	Fri 14/02/03
15		Working with Linux in a good level	30 days	Sun 26/01/03	Mon 24/02/03
16		Write code	20 days	Tue 25/02/03	Sun 16/03/03
17		Write report	20 days	Tue 25/02/03	Sun 16/03/03
18		Working with I/O card architecture	12 days	Mon 17/03/03	Fri 28/03/03
19		⊟ **3rd Stage (Milestone III)**	**42 days**	**Sat 29/03/03**	**Fri 09/05/03**
20		Combine all the previous research	15 days	Sat 29/03/03	Sat 12/04/03
21		Combine Write final code	10 days	Sat 29/03/03	Mon 07/04/03
22		Testing and error checking	8 days	Sun 13/04/03	Sun 20/04/03
23		Quality review - Correcting mistakes	5 days	Tue 08/04/03	Sat 12/04/03
24		Write Final Report	15 days	Mon 21/04/03	Mon 05/05/03
25		Review - Combine - Sign off work	2 days	Tue 06/05/03	Wed 07/05/03
26		Slag - Supervisor meetings	2 days	Thu 08/05/03	Fri 09/05/03

Appendix 2 – User Space program (source code final version)

```
/****************************************************************************
    | user.c |  -  User-Space Application for the module (labcard-alt.c)
                   which is a Device Driver for ISA I/O card PCL731
                             -------------------
    begin                : 01-Nov-2002
    copyright            : (C) 2002 by Grigorios Fragkos
    email                : djgreg@linuxmail.org

****************************************************************************/

 /**************************************************************************
 *                                                                         *
 *  This program is free software; you can redistribute it and/or modify  *
 *  it under the terms of the GNU General Public License as published by  *
 *  the Free Software Foundation; either version 2 of the License, or     *
 *  (at your option) any later version.                                   *
 *                                                                         *
 **************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <linux/errno.h>

#ifndef TRUE
# define TRUE 1
#endif

#ifndef FALSE
# define FALSE 0
#endif


int read__ (int fd);
int write__ (int fd);



int main(int argc, char *argv[])
{
   char *dev;
   char *device_name;
   char ar;
   int fd;
   int rcode,parameter,rtr;


   while (TRUE)
     {

       printf("\n Enter device driver name [path]\n or press [d] for default
(/dev/isa731/): ");
       scanf("%s",dev);

       if (*dev=='d')
         {
            device_name="/dev/isa731";
         }
```

```
        else if (*dev=='q' || *dev=='Q')
          {
             printf("\n..Application stopped!\n\n");
             return 0;
          }
        else
          {
             device_name=dev;
          }

        //file descriptor
        fd  = open (device_name , O_RDWR);

        //if device doesn't exist print an error message
        if (fd == -1)
          {

             printf ("\n Error: cannot open device or device does not exist\n
...maybe a wrong device name!\n\n");
             return -EBUSY;
          }

         rtr=getc(stdin);

        parameter=FALSE;
        while (!(parameter))
          {
             printf (" Read or Write ? [R/W]: ");
             scanf ("%c",&ar);

             if ((ar == 'r') || (ar == 'R'))
               {
                 rcode = read__(fd);
             parameter=TRUE;
               }
             else if ((ar == 'w') || (ar == 'W'))
               {
                 rcode = write__(fd);
                 parameter=TRUE;
               }
             else
               {
                 printf(" Wrong parameter!\n");
                 parameter=FALSE;
               }
          }
        rtr=close (fd);
      }
        return rtr;
}//end of main



    int read__ (int fd)//read a byte from the opened file
      {
        unsigned char byte;

        read(fd,&byte,1);
        printf ("\n Readen : %d\n",byte);
        return 0;
      }

    int write__ (int fd)//write a byte,that the user enters, to the opened
file
      {
        unsigned char byte;
        int panos;
```

```
  printf (" Enter a byte to write to device (Dec): ");
  scanf ("%d",&panos);
  printf (" You entered %d \n",panos);

  byte = panos;
  write(fd,&byte,1);
  return 0;
}
```

Appendix 3 – Kernel Module (source code final version)

```
/***************************************************************************
         | labcard-alt.c |  -  Device Driver for ISA I/O card PCL731
                             -------------------
    begin                : 01-Nov-2002
    copyright            : (C) 2002 by Grigorios Fragkos
    email                : djgreg@linuxmail.org

 ***************************************************************************/

/***************************************************************************
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 ***************************************************************************/


#ifndef __KERNEL__
#  define __KERNEL__
#endif

#ifndef MODULE
# define MODULE
#endif

#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/errno.h> //less /usr/src/linux/include/asm/errno.h
#include <asm/uaccess.h>
#include <linux/param.h>
#include <linux/ioport.h>
#include <asm/io.h>

#define __NO_VERSION__

EXPORT_NO_SYMBOLS;
//declare device name
char device_name[] = "isa731";
//char *buff;
int MajorNum = 42;
int base_port = 0x200;
int port_range = 8;
int mod_byte1 = 0x90;
int mod_byte2 = 0x80;

//dymanic name declaration @ module load time
MODULE_PARM (device_name,"s");
MODULE_PARM (MajorNum,"i");
MODULE_PARM (base_port,"i");
MODULE_PARM (mod_byte1,"i");
MODULE_PARM (mod_byte2,"i");


//Declaration of functions
static struct file_operations  FileOps;
static int __init start (void);
static void __exit clean (void);
static int register_device (char *device_name, struct file_operations
*FileOps);
static int unregister_device (char *device_name);
static int open_card (struct inode *inode, struct file *file);
static int close_card (struct inode *inode, struct file *file);
static int write_card (struct file *filp, const char *ptr, size_t count,
loff_t *f_pos  );
static int read_card (struct file *filp, char *ptr, size_t count, loff_t
*f_pos);
static int allocate_ports (unsigned int base_port, unsigned int port_range);
static void set_operating_mode();

module_init (start);
module_exit (clean);



// File Operations structure  * * * * * * * * * * * * * * * *
static struct file_operations  FileOps =
{
   .release = close_card,
     .open = open_card,
     .read = read_card,
     .write = write_card,
};

// My INIT function * * * * * * * * * * * * * * * * * * * * *
static int __init start (void)
{
   int check;

   check = register_device (device_name, &FileOps);

   if (check<0)
     {
      printk ("<1>Can't register Device :( \n");
      return check;
     }

   check=allocate_ports(base_port,port_range);

   if (check<0)
     {
      printk ("<1>Can't load Module :( \n");
     }

   else
     {
      printk ("<1>Module loaded succesfully :) \n");
     }

   set_operating_mode();

   return check;
}

//My EXIT function * * * * * * * * * * * * * * * * * * * * * *
static void __exit clean (void)
{
   unregister_device(device_name);

   release_region(base_port,port_range); //release ports
}


//Set Orerating Mode for ISA I/O card
//PortA0 will be an OUTPUT
//PortA1 will be an INPUT
```

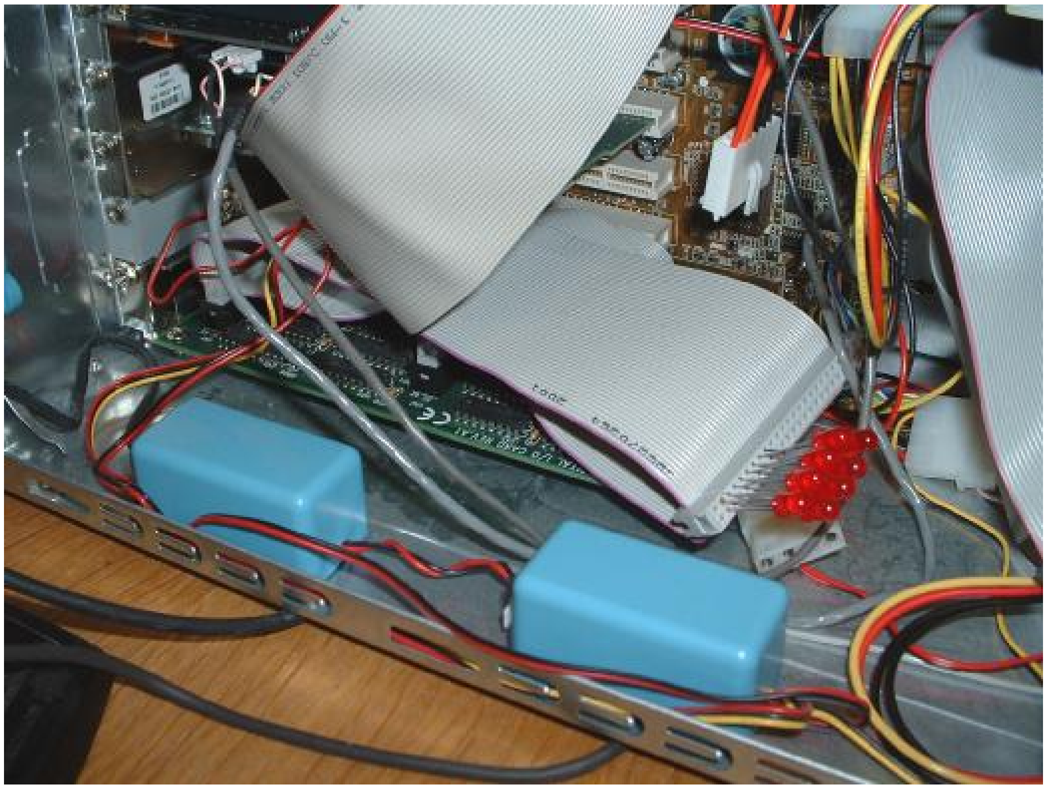
```
static void set_operating_mode()
{

   int register_address1 = base_port + 3;
   int register_address2 = base_port + 7;

   outb_p(mod_byte2, register_address1);
   outb_p(mod_byte1, register_address2);

   printk ("<1>SETOPMOD \n");

}


//Allocating Ports to the Registered Device * * * * * * * * * * * * * * * *
static int allocate_ports (unsigned int base_port, unsigned int port_range)
{

  int err;

   if ((err=check_region (base_port,port_range)) < 0)
     return err; // device busy

   request_region (base_port,port_range,device_name);

   return 0;

}



// REGISTER DEVICE * * * * * * * * * * * * * * * * * * * * * *
static int register_device (char *device_name, struct file_operations
*FileOps)
{
   int check;

   if (device_name == NULL)
     device_name = "isa731";

   //register device with kernel and return >=0 for true || <=0 for false
   //use the current Major Number for the specific device according
   //the File Operations
   check = register_chrdev(MajorNum, device_name, FileOps);

   if (check<0)
     {
      printk ("<1>Can't register device :( \n");
      return check;
     }
   return 0;
}

// UNREGISTER DEVICE * * * * * * * * * * * * * * * * * * * * * *
static int unregister_device (char *device_name)
{

   int check;
   check = unregister_chrdev(MajorNum, device_name);

   if (check<0)
     {
      printk ("<1>Can't unregister device.... Maybe in use...\n");
      return check;
     }
   return 0;
}
```

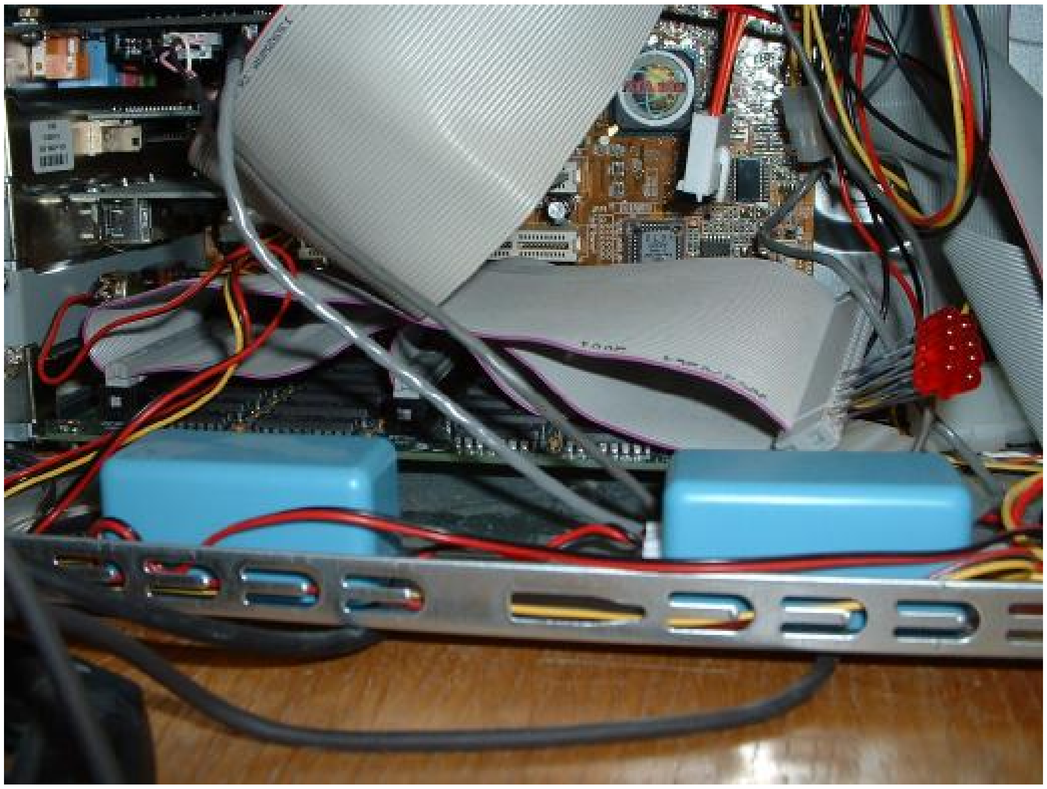
```
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
// FILE OPERATIONS
// OPEN
// READ
// WRITE
// RELEASE
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
//
static int open_card (struct inode *inode, struct file *file)
{
   printk ("<1>OPEN \n");
   if(MOD_IN_USE>0)
     return -EBUSY;  //if the card is beeing accesed already return
                     //"Device or Resource Busy"
   //increment usage count
   MOD_INC_USE_COUNT;
   printk ("<1>OPEN2 \n");
   return 0;
}


static int close_card (struct inode *inode, struct file *file)
{

   //decrement usage count
   MOD_DEC_USE_COUNT;
   printk ("<1>CLOSE \n");
   return 0;
}


static int write_card (struct file *filp, const char *ptr, size_t count,
loff_t *f_pos  )
{
   int address = base_port;
   int buff = *ptr;

   outb(buff, address );   //write to device

   return 0;

}


static int read_card (struct file *filp, char *ptr, size_t count, loff_t
*f_pos)
{
   //unsigned char address =  base_port + 4;   // * * * * * * *
   int n = count;
   while (n--)
     {
      printk ("<1>READ \n");
      *(ptr++) = inb(0x200);  // get data from device
       printk ("<1>READEN %d from device\n",*ptr);
     }

   return (int)count;
}
```

Appendix 4 – MS – DOS application (source code final version)

```
/*************************************************
* This demo program shows how to use the PCL-731's *
* readback function to monitor the output status.  *
*                                                  *
* Hardware setting:                                *
* 1. Base address set at 0x200                     *
**************************************************/

#include <stdio.h>
#include <conio.h>
#include <process.h>
#include <dos.h>
main()
{
int base = 0x200; /* set base address to 200 (hex) */
int portA; /* save readback value of port_A1 */
int portB; /* save readback value of port_B1 */
int portC; /* save readback value of port_C1 */
int i,j;

/* handle screen */
clrscr();
gotoxy(30,3);
textattr(0x70);
cputs("PCL-731 DEMO PROGRAM");
gotoxy(11,6);
printf("PortA0 output value -> ");
gotoxy(11,8);
printf("PortB0 output value -> ");
gotoxy(11,10);
printf("PortC0 output value -> ");
gotoxy(43,6);
printf("PortAl Readback -> ");
gotoxy(43,8);
printf("portB1 Readback -> ");
gotoxy(43,10);
printf("portC1 Readback -> ");

/* initialization */
outportb (base+3,0x80); /* set 8255 port0 all as output */

for ( j=0;j<0x100;j++ )
{
      outportb (base,j); /* out j to port AO */
      gotoxy (34,6);
      printf("%2x",j);
      portA = inportb (base);
      gotoxy(63,6);
      printf("%2x",portA);
      if ( portA != j )
      {
            printf ("\7"); /* beep */
            gotoxy (30,13);
            textattr(0x09);
            cprintf("PortAl readback error!");
          getch();
          //exit(1); /* quit to dos */
      }

      outportb (base+1,j); /* out j to port BO */
      gotoxy (34,8);
      printf("%2x",j);
      portB = inportb (base+1);
      gotoxy(63,8);
      printf("%2x",portB);
      if ( portB != j )
      {
            printf ("\7"); /* beep */
            gotoxy (30,13);
            textattr(0x09);
            cprintf("PortB1 readback error!");
            getch();
            //exit(1); /* quit to dos */
      }

      outportb (base+2,j); /* out j to port CO */
      gotoxy (34,10);
      printf("%2x",j);
      portC = inportb (base+2);
      gotoxy(63,10);
      printf("%2x",portC);
      if ( portC != j )
      {
            printf ("\7"); /* beep */
            gotoxy (30,13);
            textattr(0x09);
            cprintf("PortC1 readback error!");
            getch();
            //exit(1); /* quit to dos */
      }
} /* end of for */
} /* end of main() */
```

Appendix 5 – The ISA I/O card PCL731





1st Sub-Report

About the report:

This is the first sub-report of the project. According to the students handbook this must be a sub report detailing the research undertaken in the first term of study. The research for the project "Linux Device Drivers" consist of the following areas: Familiarization with the GNU/Linux operating system, kernel module programming, and understanding the specifications of an I/O ISA card.
The distribution of Slackware 8.1 and the 2.4.18 official Linux kernel were used for the familiarization with GNU/Linux operating system.

Introduction:

Linux is an operating system that is spreading fast into the computing market. We can consider of Linux as the rising star amongst operation systems. A major question must be answered before we continue, why people like Linux? The other operating systems have a lot of limitations, they are designed for low-end home users and do not deliver the performance or flexibility that we expect. Furthermore Linux is free, stable, and runs on the majority of processors like Pentium Pro, Pentium II, Pentium III, Pentium IV, AMD, Cyrix chips and even older 386/486 machines as well as some other platforms. In fact Linux can be ported for almost every platform, even for R.T.E.C.S[14]. At this point we should mention that Linux is a redistributable clone of the UNIX operating system which it uses the X window system graphical user interface (GUI) and offers several different integrated desktop environments such as KDE and GNOME. Despite the fact that Linux supports a great majority of devices, a great part of the manufacturing companies, that produce computer-related hardware, do not support their products with Linux device drivers yet. Subsequently, development of those drivers is needed to be done by the users themselves.

[14] Real Time Embedded Computer Systems

The Kernel of the operating system is a bunch of code that acts as an interface between the application layer and the hardware. The application layer is the user space environment where all the applications are executed by the user. In order for the user-space program to communicate with the hardware from the application layer, the interception of a kernel is vital. The kernel looks for the particular driver for the device we want to access and, according to the instructions on that driver, will allow the program to communicate with the specific piece of hardware (e.g. Hard Drive, CD-Rom, ISA card). This driver is a module for the kernel usually written in C programming language. Actually the most of the kernel is made up from too many kernel modules linked together.

The Peripheral Component Interconnect (PCI) bus allows us to connect a hardware card into our computer. An example of a PCI card is the well-known soundcard we have in our computers. Nowadays, the PCI standard as a result of its cost effectiveness, backward compatibility, scalability, and forward-thinking design, has overcome ISA (Industry Standard Architecture) standard which is the ancestor of the PCI bus. Even thought, the ISA bus architecture is still used by most manufacturers; this is due to the fact that the transaction from the one architecture to the other is very expensive for the most of the industry and the managers think that is this is not a value for money solution.

The object of this project is to familiarize the student with the concepts of Linux Kernel programming, the GNU/Linux Operating System, ISA bus interface and C programming, in order to develop, implement and test a kernel module which will provide the appropriate environment to a user-space application to access the ISA I/O card. [Reference a2, a4]

Linux:

"*Linux is a UNIX clone written from scratch by Linus Torvalds in 1992 with assistance from a team of developers across the network.*"[15] It was first developed for 386/486-based PCs. These days it also runs in several other machines like DEC Alphas, SUN Sparcs, M68000 machines (like Atari and Amiga), MIPS and PowerPC. During this 10 years, that Linux exists, it has become a very wide-use operating system, mostly used by well experienced users and for server or embedded computers use.

However, it still can't overcome the Windows® monopoly on the market. Therefore, most device manufacturers, for the above platforms, do not make any drivers to support their products for the Linux environment. This leaves, the Linux users, with two options; either not buy the product or build the drivers by themselves. C is a function oriented programming language, which was devised in the early 1970s as a system implementation language for the nascent UNIX operating system. Nowadays, it has become one of the dominant languages. The Linux kernel and most of its modules were built in C platform. By 1973 UNIX OS was almost totally written in C. Thus, in case that someone needs to update the kernel or implement a device driver, C language is the best approximation to do that. [Reference a2, a4]

Familiarization with Linux:

Linux has many distributions; the most popular are Red Hat, SuSe, Slackware and Debian. Usually you choose the right distribution for the right job. In the following pages we will explore the world of Linux through the Slackware 8.1 distribution which uses the 2.4.18 official Linux kernel. The first thing we must describe is how to install Slackware into our system.

[15] http://mirror.ati.com/support/faq/linux.html

Considering that the most users have Windows® installed into their system we must say that those two operating systems can co-exist into the same operation system, for that reason there is no need of uninstalling Windows® in order to install any of the distributions of Linux. The only thing we must be aware is that we should install Linux in a separate partition on our hard drive.

In that separate partition we should create two partitions; the first is the root partition where all the GNU files among the Linux kernel will be installed, the second is the swap partition (a partition which will be used as extra memory for the operating system). Some other distributions, like Red Hat®, choose to use an extra partition for the /boot directory, where the Linux kernel and all the booting configuration files will be stored.

Except the fdisk partition table manipulator, GNU/Linux uses some other similar, yet more user friendly, programs such as the Disk Druid and cfdisk, in order to help as partition our hard drive. Red Hat and Slackware have a user friendly interface in order to help us install GNU/Linux into our system. The file systems supported by GNU/Linux are plenty, like reiserfs, fat32®, minix®, ISO9660, udf, ext2/3 and MS-DOS®. (MP (8) mount)

Usually, we use the console environment as the basic environment which is something like MS-DOS® command line in Windows® but with furthermore capabilities and functionalities. The administrator login in Linux is called root and we navigate through the directories using the following commands (cd, cd .., cd /, ls etc). In order to find a program or command that is suitable for a specific operation, we can search through the *apropos* (MP (1) apropos) database and then use it's manpage (MP (1) man) to understand the way it works. In order to exit a manpage press the Q key on your keyboard.

All devices in Linux have a special file to be represented. This special files are into the /dev directory and the following table have some of them. In general, everything, from a device to a memory map, in GNU/Linux is represented by a file. This means, in order to access it, we have to use the normal file operations we would use for a simple text document.

Special File Form	**Example**	**Device / Use**
/dev/fd*n*	/dev/fd0	Floppy Disk
/dev/rmt*n*	/dev/rmt0	Generic tape devices
/dev/rst*n*	/dev/rst0	SCSI tape drive
/dev/cd*n*	/dev/cd0	CD-ROM devices
/dev/tty*n*	/dev/tty01	Serial Line (hardware terminal / modem)
/dev/pts/*n*	/dev/pts/0	Pseudo-terminal (used for network sessions)
/dev/console	/dev/console	Console Device
/dev/mem	/dev/mem	Map of physical memory
/dev/kmem	/dev/kmem	Map of Kernel virtual memory
/dev/mouse	/dev/mouse	Mouse Interface
/dev/ram*n*	/dev/ram00	Ram Disk
/dev/sd/a-g/*n*	/dev/sda0	SCSI Disk
/dev/swap	/dev/swap	Swap device
/dev/null	/dev/null	Null device: output written to it is discarded

(Introduction to UNIX security, Dr. Andrew Blyth, Lecture Notes 2002, MSc Information Systems and Network Security, University of Glamorgan)

A simple PS/2 mouse will work perfectly in GNU/Linux but in case we have a wheel PS/2 mouse the following hint might be very useful for the users that has get used to working with the wheel in order to scroll through the pages of a document. Suppose we have an IntelliMouse mice. In order to make the mouse wheel to work we should open the X1186Config file with the jed editor by typing the following command. (Remember that is case sensitive)

```
jed /etc/X11/XF86Config
```

We are looking for the line that into the file that has the following command.
Option "Protocol" "PS/2" and we replace it by Option "Protocol" "IMPS/2"

After 2 lines we add the following command also.
Option "ZAxisMapping" "4 5"
We can press Ctrl + X + C in order to exit and we press y to save the changes.
If we did that through the X-Windows, we must restart our X-Server.

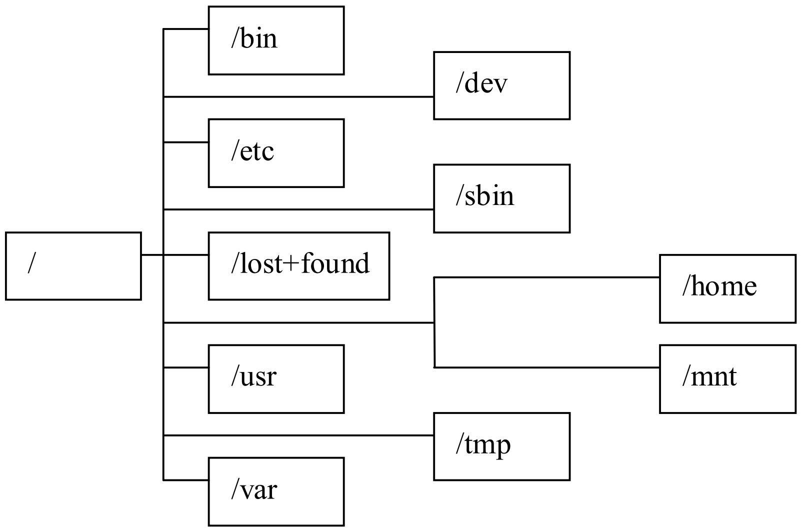
The editors that we can use in order to view or edit files in GNU/Linux are plenty. The most well-known are jed, emacs, pico, nano, vi and vim which can be used to modify files and more or less, which can be used only to view a file (read only mode).

In the following table we will see some of the most important commands and files of GNU/Linux along with a small description in order to have a generic idea of what we are going to talk about in the following pages.
Before we go along we must describe another helpful functionality of GNU/Linux. Typing the name of a folder, file or command we can "auto complete" the rest of the word by clicking the tab button. If we double click the tab button will return all the possible commands and pathnames that can be used from that user at the specific location starting with the letter(s) we've typed.

File / Command	**Description**
jed /etc/passwd	Will open the passwords file into the jed editor
less /etc/shadow	Will open the shadow file into the less editor
ls | less	Will send the result of the ls command into the less editor
ls > /greg.txt	We can save to a file, called greg.txt in the top level directory, the result of the ls command
su	Usually in GNU/Linux we log in as simple user for security

	purposes. In order to change our login to super user (root) we must type su (without log off and log on again) and the system will prompt as for the root's password
su user	Change to "user" login
/usr/sbin/adduser	In order to add a new user to our system
/usr/bin/passwd	In order to change, modify passwords
pwd	This will return the current path that we are working on
ntsysv	Adjustments of what is going to start at boot time (RedHat)
ps -adl	This will display the processes currently running detailed
export PS1=	In order to change the prompt e.g. export PS1="Yes Master"
who	Lists all users currently logged in to the this machine
whoami	Display information for the current user login
id	Display id number and privileges of the current user login
find /-name <filename>	Searches the /file system for the requested <filename>. Wild characters such as * and ? can be used.
whereis <command>	Displays in which directories, included in the $PATH, the <command> is situated.
echo $PATH	Displays the path directories
/usr/bin/pine	A program that will allow as to access our e-mails
/usr/bin/elm	A program that will allow as to access our e-mails
/sbin/ifconfig	It's the same command as ipconfig for Windows ®
ifconfig eth0 up	Enable the first Ethernet card. The "ifconfig eth0 down" will disable it
/sbin/netconfig	Network configuration
/sbin/insmod module.o	Link a module with the Kernel
/sbin/rmmod module	Unload a module from the Kernel
/sbin/lsmod	We can see the modules that already have been loaded
/usr/sbin/cfdisk	Similar to fdisk partition table manager of Windows ®
less /etc/fstab	Which devices will be mount by themselves on GNU/Linux boot time. e.g root must be automount
less bin/dmesg	In that file we can see the outputs messages of the Kernel
whatis <command>	Displays a minimum help file for the command

In the diagram below we can see the directory structure in GNU/Linux. The top directory as we've already mention is called root. The following diagram along with a part from the description of the directories is from *(Introduction to UNIX security, Dr. Andrew Blyth, Lecture Notes 2002, MSc Information Systems and Network Security, University of Glamorgan)*



Starting from the top in the previous diagram we can see the **/bin** and **/sbin** directory which is the location for the binary files and/or script files. The **/dev** is the device directory containing special files that allow to the operating system to reference each piece of hardware. The **/etc** directory contains system configuration files and executables. A very important and useful directory is that which called **/lost+found**. Disk errors or incorrect system shutdown may cause files to become lost. Lost files refer to disk locations that are marked as in use in the data structures on the disk, but are not listed in any directory. The **/home** directory is a conventional location for user's home directories. The **/mnt** directory is a directory implemented in order for the users to mount some media, like our CD-ROM drive or a hard drive etc, into their file system. We will explain more about the /mnt directory in the following page.

The following directory is the **/usr**, which, in the old days, was used to store personal configuration files for each user. Nowadays, it's initial usage has been replaced by the /home directory. It's current use is to store configuration files for most of the applications. The **/tmp** directory is available to all users and programs as a scratch directory. Last is the **/var** directory which we can think as the program files directory of Windows ®.

In order to access a device like the CD-Rom drive we must first "mount" the device.
That means we must allocate a directory that will be a link to the filesystem of the specific device. Our devices are in the "/dev" folder. If for example, our first hard disk drive is called "hda", the second "hdb" etc. then in order to access the second partition of our 1st hard drive we must type "hda2".
Of course the floppy disk drive is "fd0" and if we have a second one it is called "fd1" (the devices are explained in the table at page 6).
The following command will connect the device "fd0" (1st floppy disk drive)
to the "floppy" directory inside the "mnt" directory.
"mount /dev/fd0 /mnt/floppy"

To disconnect the device we must type "umount /dev/fd0" or "umount /mnt/floppy"

We could use "mnt" directory to mount a device. If we do that the "mnt" directory will be our device now until we unmount it. When we unmount the "mnt" directory we will be able again to see the contents of the original directory.

We can think of mount operation as a stack. We can mount several devices, one above the other, on a single directory. The last device we mounted (e.g. CD-ROM) is the device we are going to use from that directory. Unmounting the last device, automatically the directory is mounted to the previous device we had mounted (e.g. Floppy Drive). We will continue unmounting devices until we get the original directory "mnt". This is not useful very often, and thats why we use different directories for different devices. (MP (8) mount) [Reference a2, a3, a4, a5, a8]

Shell Scripting

The shell scripting in Linux is similar to vbscript of windows and is very easy to use in order to implement rapid scripts. At the Appendix B there is a simple example that shows some useful commands in shell scripting. After writing our script we must change the privileges of the file in order to make it executable and be able to run it. A visual representation of the privileges of a file is showed below.

```
  r w x r w x r w x
 -|-  -  -|-  -  -|-  -  -|
```

r - Stands for read
w - Stands for write
x - Stands for execute

We can see the privileges of a file typing the command

ls -l

As we can see there are three triplets of rwx. The first triplet is the owner's privileges. The second triplet is the privileges of the owner's group and the third triplet is the other groups' triplets. Assigning a number for each letter e.g. r -> 4, w -> 2, x -> 1 we can create a unique number for each privileges we want to give. If we want read, write and execute for the owner we add 4, 2 and 1 which gives us 7. If we want read only access for the other groups' triplets we use the number 4. In our case we came up with the number 764.

```
  r w x r w x r w x
 -|-  -  -|-  -  -|-  -  -|
  |4 2  1|4 2 1|4 2 1|
   \     / \     / \     /
     7        6     4
```

We must type the following command to change privileges

chmod 764 file

in order to run the script we have just created we will type

```
./myFile
```

or we can move it into a directory that is included in the path and just type

```
myFile
```

[Reference a5]

Kernel:

Init runlevels

From the moment that an operating system starts booting until it reaches the run level we wish, it passes through the init faces. This faces are the init run levels which are 6 from init 0 to init 6. Below we can see what is happening to the operating system as it passes from the one stage (run level) to the other.

init 0 - (init 0 and init 6 are the same thing)

init 1 - single user (not multi-user), (no network)

init 2 - always empty (in order to develop a custom run level)

init 3 - multi-user (without x) plus network

init 4 - Depends of the distribution (maybe either empty or represent multi-user, graphical environment with networking)

init 5 - Depends of the distribution (maybe either empty or represent multi-user, graphical environment with networking)

init 6 - Reboots the machine.

Recompile your kernel

In order to use some hardware devices that we might have, or to be able to "see" the ntfs file system of Windows ® from GNU/Linux we must recompile our kernel. Actually that recompile will add into the kernel or link to the kernel modules that are necessary in order to be able to perform the above operations. These modules are either drivers for the hardware devices or pieces of kernel code in purpose to do a specific task.

In the following pages it will be explained as better as we can how to recompile the kernel.

First of all, we have to make sure that linux's source code is installed into our system. If we are running an rpm based distribution (such as Red Hat) and the kernel source package is linux-source.rpm, we can type

```
rpm –v linux-source.rpm
```

Another way, used on every distribution, to check if we have the kernel source installed, would be to change into the kernel source directory using

```
cd /usr/src/linux
```

and check if the source code is there.

As soon as we verify that we have the source code, we can do a

```
make clean
```

to remove any temporary and unwanted files from that directory. Then we can run one of the three *.config*[16] editors by typing one of the above command lines:

```
 make config
```

OR

```
make menuconfig
```

OR

```
make xconfig (only for X)
```

The *.config* file is a configuration file, which we can see by typing

```
less ./config
```

lining in the /usr/src/linux directory and it stores every information the "make" program needs to know about which kernel modules to compile with the kernel and which to compile as loadable modules. After editing this file, we can save the new kernel configuration and check if there any modules included in the kernel that are depended from some others which were not included with

```
make dep
```

If the previous command is successful we can move into compiling a compressed image of our new kernel by typing

```
make bzImage
```

and then compiling and installing the modules with

[16] files starting with a dot [.] are hidden and can be seen with the ls –l command

```
make modules && make modules_install
```

Finally, if everything has gone right during the compiling and installing process, we can add the new kernel image "bzImage", situated into /usr/src/linuc/arc/i386/boot/, into our boot loader (such as lilo or grub) and then reboot into it. [Reference a4, a5]

The LILO boot loader

One of the most widespread boot loaders, for Linux partitions, is LILO. Lilo reads the configuration file /etc/lilo.conf and then stores the boot parameters in the first 512 bytes of a partition or in a hard drive's MBR (Master Boot Record). When a PC boots, it reads off those 512 bytes and then loads the kernel image described there. In order to change the configuration file we can either run

```
liloconfig
```

or manually edit it by typing the following command

```
jed /etc/lilo.conf
```

In order to add another kernel image option for example /bzImage we have to declare some variables (the author places them at the end of file). First of all, the image's path. This can be done by typing into the lilo.conf

```
image= /bzImage
```

Then the root partition should be stated with

```
 root= /dev/hda3
```

Additionally, we must assign a name to the new kernel image, in order to recognize it at boot time in the lilo prompt.

```
 label= Slackware.new
```

Finally, because Linux at boot time runs some diagnostic tools on the root partition, in order to prevent any damage to the file system, we initially mount it as read only. So, we add the following command.

```
 read-only
```

If we want lilo to be installed on the MBR the following line should exist on the top of the file.

```
#...
boot= /dev/hda
#..
```

Moreover, in order to have a time period equal to 50 ms in order for the user to be able to choose which kernel to boot we must add the following lines in the lilo.conf file.

```
prompt
delay = 50
```

As a final point, we have to type the command

```
lilo
```

in the command line.

[Reference a2, a4]

Kernel programming

As it is already mentioned in the introduction, most of the kernel source code is written in C programming language. The best approach, by the author's point of view, to develop a Linux kernel module is by using C.

When programming for the kernel, the standards of ANSI[17] C still exist. However, there are some differences between kernel space and user space programming. There are some libraries used only to be linked with kernel modules and consequently there are some functions that will only be recognized by the kernel.

First of all, we have to go over the structure of standard kernel module. Each module should have one "init" function and one "exit" function. Each time the module is loaded the kernel executes the "init" function, which usually checks for the availability of some resources and/or registers some of them. When the module is unloaded the "exit" (cleanup) function is executed in order to un-register everything that was previously registered by the module and leave the kernel as it was before. If the "init" function fails (returns a minus integer number) the module is considered not to be loaded. On the other hand, a cleanup function should never fail, that's why the programmer should make sure that every procedure included in this function must be successful.

As everything in the GNU/Linux operating system is treated as a file by the kernel, then a file operations structure should exist in order to declare which operations are permitted and how the kernel should react when each of them is called by the user space. Whenever file operation is performed on a file by the application layer, the kernel searches for the module that has been register with this specific file and responds as its file operations structure describes.

[17] American National Standard Institute

After the C program has been developed, it must be compiled (but not linked) using a ANSI C compiler. Using the gcc compiler this can be done with the following command line

```
gcc  -c module-name.c
```

This will produce our ELF binary *module-name.o*. This should be linked then into the kernel by typing

```
insmod module-name.o
```

Now the module should be loaded into the kernel. We can verify that by listing all the modules loaded into the kernel with

```
lsmod
```

and checking if it is in the listing.
If someone would like to unload the module, he should type

```
rmmod module-name
```

[Reference a1, a3, a5, a6, a9, a10]

ISA Input Output Card

A BUS, in the computer world, is defined as the medium around which the data from a device to another one circulate or of a device to the memory and/or the CPU. A card can communicate with the CPU using a rank of ports, possible interrupts and DMA(s)[18]. In the following points, it is possible to be seen as they are assigned or not in each type of bus architecture.

[18] Direct Memory Access

The first bus architecture that was implemented in PC architecture was ISA. At first it was an 8-bit medium but it quickly evolved in 32-bits. This indicates that every time, in each clock cycle, the bus was able to transfer one and two bytes respectively. Even thought technology enables to increase the bus frequency for ISA, backward compatibly issues force it to be stable at 8.33 MHz. A 16-bit ISA card is able to support a maximum of 16 megabytes of transference per second for all the devices simultaneously, a record which has been overcome long time ago.

The PCI architecture is the newest and most widespread, now days, bus architecture. Norm PCI, indicates that a 4 BUS PCI is solely of 4 slots, but allows us to connect another 4 slots in groups, by means of "PCI bridging". Also, PCI is a 32-bit bus with a frequency of 33 MHz. This means that the maximum transfer rate can be 133 megabytes per second. This is more than double the maximum speed of an ISA bus. Additionally, IRQ[19] sharing can be supported by this architecture, something that PnP[20] and Windows® take a great advantage of.

The author initially had chosen to use a PCI card for the purposes of this project. Unfortunately, a posting error forced the use of an ISA Input Output card, whose specifications will follow.

The exact model code of the card is PCL-731, which is a 48-bit Digital Input Output (DIO) card. This means that it is equipped with 48 pins, which can be used as inputs or outputs to and from the card. Several configurations can be made in order to set some pins as inputs and some others as outputs. In addition, a "8250 family" microcontroller is attached to the card and is responsible for the algorithm of the digital signal processing of the data coming into and going out of the card. This microcontroller is fully programmable and can change completely the functionality of

[19] **I**nterrupt **Req**uest Line
[20] Plug and Play

the card. Still, the two IRQ's , of the card ,as well the trigger level of the them can be manually selected with the use of two jumpers.

Furthermore, the card has two 50-pin OPTO-22 compatible connectors, where the 48 input/outputs are situated, divided into six 8-bit ports I/O ports, along with some registers. A 50-way IDC connector may be easily connected to them and then transfer the data into an IDC cable connected to an external device. Also, the card is equipped with 48 LED's[21], corresponding to the 48 I/O pins, showing the high or low logic for each pin. The logic High voltage for each input varies between 2.0 and 5.25V and the logic Low voltage for the previous is above 0.0 and below 0.8V. The logic High output signal is 2.4V and the logic Low of the same output is 0.4V.

As it was mentioned before, the 48 I/O pins are divided into six 1-byte ports. Each one of them is assigned to a port address always in respect to the base address of the card which can be in the range from 0x0 to 0x3F8. Below a table of all the I/O ports and their corresponding address is shown.

Address	**Port**
Base + 0	**AO**
Base + 1	**B0**
Base + 2	**C0**
Base + 3	**CFG REG**
Base + 4	**A1**
Base + 5	**B1**
Base + 6	**C1**
Base + 7	**CFG REG**

[Reference a5, a11, a12, a13, a14]

[21] Light Emitting Diodes

References:

[a1]Daniel P. Bovet, Marco Cesati 2000. *Understanding the Linux Kernel. O'Reilly.*

[a2]Bill Ball 1998. *Teach Yourself Linux in 24 Hours: Second Edition. Sams Publishing.*

[a3]Alessandro Rubini & Jonathan Corbet 2001. *Linux Device Drivers: Second Edition. O'Reilly.*

[a4]Matt Welsh, Matthias Kalle Dalheimer, Lar Kaufman 1999. *Running Linux. O'Reilly.*

[a5]Richard Stones & Neil Matthew 2001. *Beginning Linux Programming.Wrox.*

[a6]Jesse Liberty, David Hovarth 2000. *Teach Yourself C++ for Linux in 21 Days. Sams.*

[a7]Don Anderson, Tom Shanley 1995. *ISA System Architecture. Addison Wesley*

[a8]*Introduction to UNIX security*, Dr. Andrew Blyth, Lecture Notes 2002, MSc Information Systems and Network Security, University of Glamorgan.

[a9]Linux Journal, October 2001, issue 90, How to write a Linux USB device driver p.24.

[a10]Linux Magazine, September 2002, issue 23, language of C p.66.

[a11]Advantech. User's Manual for the ISA card PCL-731. 48-bit Digital I/O card.

[a12]http://www.cse.ogi.edu/class/cse521/2002fall/Lec02-ISA.ppt

[a13]http://sunsite.tut.fi/hwb/co_ISA_Tech.html

[a14]http://www.cs.arizona.edu/computer.help/policy/DIGITAL_unix/AA-Q0R6C-TET1_html/TITLE.html

Appendix A

Hello_kernel.c

```
#include <linux/kernel.h>

#define __KERNEL__      //define that this part of the source code of the kernel
#define __MODULE__      //define that this is a kernel module
#define __NO_VERSION__    //inform the compiler that the module will be kernel
                          //version independent


module_init (hello)   // define the init function of the module
module_exit (bye)   // define the exit function of the module

static int __init hello(void)
{
   printk ("<1> Hello from kernel!!!\n");  //the kernel prints a message
   return 0;
}

static void __exit bye(void)
{
   printk ("<1> Bye bye from Kernel!!!\n");
}

/* The number <#> into the printk statement declares the priority of this kernel
message among the others. 1 is the highest priority */
```

Appendix B

MyFile.sh

```
#! /bin/bash
k=1
echo "running script..."
echo "mounting..."
mount /dev/hda /mnt
read -p "enter key.."
echo "unmounting..."
umount /dev/hda
echo $k
exit
```

Explaining the previous notation.

```
!#                      -> With whitch cell i'll run the script
#                       -> comments
echo                    -> print
echo -n "Your name: "     -> -n means no new line
k=getwchar()
read  -p "Your name: " K  -> read input and place it into K
```

2nd Sub-Report

About the report:

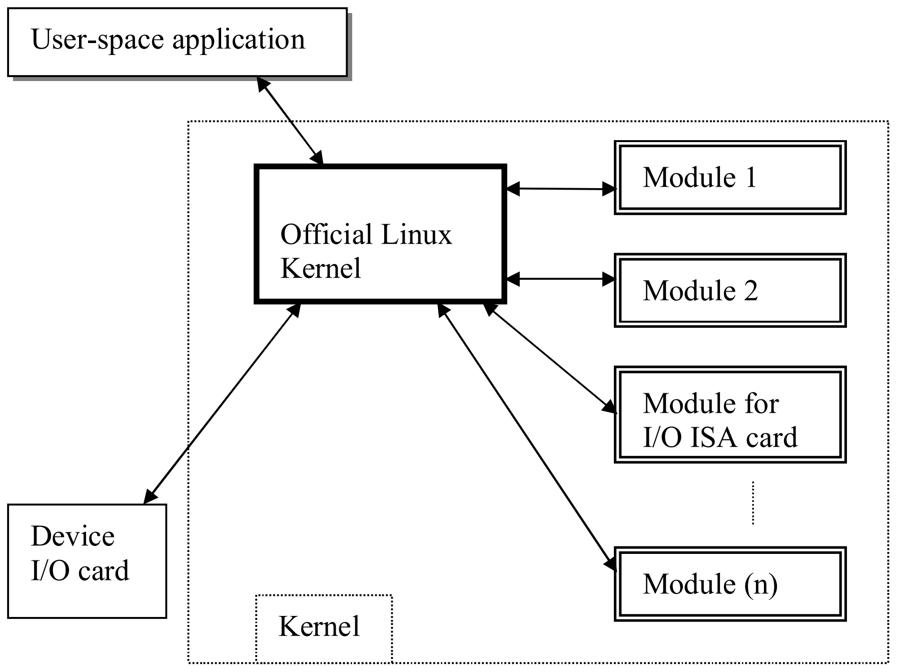
This is the second sub-report of the project. According to the students handbook this must be a sub-report detailing how the research has been evaluated and applied with respect to the design and development of the proposed system. This report will not normally include details concerning any stage of the actual coding. Because of the nature of the current project and the requirements of the supervisor this sub-report will include the source code of the project and how that was achieved. The main research that was undertaken during that stage of development was to design, understand and develop the module for the I/O ISA card.

Introduction:

In the following lines of the report we are going to cover the major aspects concerning a device module implementation. We will start approaching the kernel structure and how it's handles the modules in order to access devices. Furthermore, we will see how a module links to the kernel and how it performs the various operations. Additionally, we will explain the role of the user-space program in order to access the device. In order to make use of the I/O card we will analyze how the ISA card was programmed using the specification of the user's manual. The issues concerning the C code that was produced and the source code structure will be covered. Finally, some additional information concerning an easier way to manipulate the module by writing a small script and a section about interrupts is provided.

How a module works:

In the following diagram we can see how a module is linked to the kernel and how it manipulates requests from the user-space program or the device through the kernel in order to perform the requested operations.



Once we have developed a module we must link it to the kernel. Using the insmod command and the name of the ELF[22] file we can load our module to the Kernel (e.g. insmod isa731.o). Typing the lsmod command we can see if the module was successfully loaded or we can use the dmesg command to see what messages the module sends to the kernel. When a module is linked to the kernel, it registers a range of ports and a major number. This range of ports is going to be accessed only by our module and that's where the device is connected. The major number is a unique number in order for the kernel to identify the specific device. The module doesn't

[22] (Executable and Linking Format)

know the device's representative filename in the file system, but just its major number. Any device with the same major number is manipulated by the same module. Then, they are distinguished by a second numerical prefix, which is the minor number. The user-space program, assuming that it is trying to perform an operation using the I/O hardware device, sends to the kernel a request in order to open the specific device. The kernel is looking for the specific module that is loaded for that specific device, matching the major number from the device and the module. The module holds information about what the kernel should do when a request reaches it. When the request arrives to the kernel it searches for the appropriate module to manipulate the request. The module "tells" to the kernel what to do with the particular request and then the kernel "tells" to the device to do it. Finally, the kernel will return to the user-space program any reply that comes from the device, if any. The same operation takes place when a request comes from the device in the opposite direction. [Reference b1, b3, b5]

Devices

Linux has a way to identify the type of devices that correspond to the system. These devices, which normally exist into the /dev directory, can be accessed by modules that consist of three different classes. These three classes of modules are the char modules, block modules and network modules. Every module that is included into the kernel or a module that we might want to load manually must have this type of information. A small description is provided in the following lines in order to understand how the system recognizes each of the three classes of modules.

A character device can be accessed as a stream of bytes. This behavior is implemented by a char device. This driver usually implements four types of system calls, the open, close, read and write system calls. Char devices are accesses by means of file system nodes[23]. A char device differs from a regular file just in the fact that in the regular file you are always able to move back and forth, while char devices are just data channels. This means that you can only access sequentially.

[23] E.g. /dev/tty1 and /dev/lp0

A block device, as well as char devices are accessed by file system nodes in the /dev directory. A block device is able to host a file system, for example a disk. Linux permits the application to read and write a block device like a char device, and allows the transfer of any number of bytes at time. Consequently, block and char devices are different just in the way the kernel manages internally the data and therefore in the kernel/driver software interface. The differences between them are noticeable to the user. A block driver gives the kernel the same interface with a char driver. Furthermore, it offers an additional block-oriented interface which is invisible to the user. In addition it can offer applications opening the /dev entry points. However this interface is crucial to have the ability to mount a file system.

Every network transaction is made through an interface. This means that there is a device that can exchange data with other hosts. In most cases an interface is a hardware device; however it is possible to be a pure software device[24]. The role of the network interface is to send and receive data packers, driven by the network subsystem of the kernel, with no knowledge of how individual transactions map to the actual packets being transmitted. A network interface is not easily mapped to a node in the file system due to the fact that it is not a stream-oriented device. The Linux way to supply access to interfaces is also by assigning a unique name to them[25]. Nevertheless that name does not have a corresponding entry in the file system. The kernel and a network device driver have a completely different communication from char and block drivers. The kernel calls functions are associated to packet transmission as a replacement for reading and writing.
[Reference b3, b4]

Device Major Number

In order for a device to be recognized by the kernel among the other devices, each device is assigned a unique number. This number is called the “device’s major number”. This can be set during the init function of the module. The programmer has the choice to either allocate it dynamically, that is let the kernel decide about the major number (kernel 2.4.x), or manually, by specifying the number in the source

[24] e.g. loopback interface
[25] e.g. eth0

code of the module or at load time. In the current design of the module the second approach was used. This is because, in order to make a node for the card in the file system, we must know the major number of the device assign during the load of the module. In case we had used the third option, (let the kernel decide the major number), we should look into the /proc/devices file in order to find the major number. Instead, the manual allocation of the major number was used and the only thing we should be aware of is that this number is not in the /usr/src/linux/Documentation/devices.txt and is not already registered to the kernel. The chosen number to be used as major number for the ISA card is 42.
[Reference b3, b5]

Device name

A unique name should be invented in order to associate it with the device. This would be the modules name as appeared at the output of the command lsmod and the node's name appeared under /dev/. The author of this project chooses the name **isa731** as a device name.
[Reference b3, b5]

Usage Count

Usage count is a variable that we change each time we using the device. This is for the reason that we don't want no other module to access the device when we are using it. Using the functions MOD_INC_USE_COUNT and MOD_DEC_USE_COUNT we can increase and decrease the usage count. This operation takes place when we use the file operations open and release. When we open the device we must increase the usage count and when we release the device we must decrease the usage count.
[Reference b3, b5]

How can we create a node

As we already mention a node is a file that represents a device into the /dev directory. In order to create a node we must use the following command before load our module to the kernel.

```
mknod DEVICE_NAME c MAJOR_NUM MINOR_NUM
```

E.g. `mknod /dev/isa731 c 42 0`

That command creates a node into the /dev directory named isa731 that is a character device (c) with major number 42 and whose minor number is 0.

The minor number can help to open the same device in a different way. Furthermore, if we have on our system 2 identical devices e.g. two ISA I/O cards, we can access them using the same module but the node for each card will have a different minor number.
[Reference b3]

Loading modules automatically

An especially useful feature is the kernel daemon “kerneld”. The kernel can load needed device drivers and other modules automatically without manual intervention from the system administrator. If the modules are not needed after a period of time (60 seconds), they are automatically unloaded as well. In order to take use of the kerneld we must turn it on during the kernel configuration along with the System V IPC option. The kernel configuration is the kernel recompile that we discussed in the first sub-repost of this project.
[Reference b3, b4]

User Space Program:

In this section of the report we would like step through and explain some basic points of the user’s space source code.

The source code of the user space is included in the appendix 1.

The user space program, in order to send or receive data from the device, must use the module as the middle man. The module in order to access the device uses the special function to write to ports. From the user space program we must use normal file operations on the devices representative file. In order to use the file operations through the module that we developed we must open the file. This, will open the device in order to be used (the open_card function of the module will be executed) or if the device does not exist it will print a message.

Then, we have a loop in order for the program to prompt us continuously if we would like to read from or write to the device. Pressing the Ctrl + C the program will exit. In each case (read or write) we call a function. These functions are read__ and write__. In the write__ function using the file descriptor, acquired from the open function, we write a value to the file. At that time the write_card function of the module is executed. The read__ function reads from the file the input value from the device and causes the read_card function of the module to execute. When the program is terminated by the software interrupt Ctrl + C, the kernel closes all the files with file descriptors used by that program and the module executes its close_card function.
[Reference b3, b5, b6]

How the ISA I/O card was programmed:

The ISA I/O card PCL-731 has two 50 pin IDC connectors. In the following picture we can see the IDC connectors. According to the user's manual of the ISA card we must register two addresses in order to use them for writing and reading from the device. The student, after some testing, decides to use the CN1 IDC connector to write to the device and the CN2 IDC connector to read from the device.

The pins 49 are the pins that provide the 5Volts. All the GND are the ground pins and the rest are the pins that we can send or read bits (logical 1 or 0).

CN1

PC07	1	2	GND
PC06	3	4	GND
PC05	5	6	GND
PC04	7	8	GND
PC03	9	10	GND
PC02	11	12	GND
PC01	13	14	GND
PC00	15	16	GND
PB07	17	18	GND
PB06	19	20	GND
PB05	21	22	GND
PB04	23	24	GND
PB03	25	26	GND
PB02	27	28	GND
PB01	29	30	GND
PB00	31	32	GND
PA07	33	34	GND
PA06	35	36	GND
PA05	37	38	GND
PA04	39	40	GND
PA03	41	42	GND
PA02	43	44	GND
PA01	45	46	GND
PA00	47	48	GND
+5 V	49	50	GND

CN2

PC17	1	2	GND
PC16	3	4	GND
PC15	5	6	GND
PC14	7	8	GND
PC13	9	10	GND
PC12	11	12	GND
PC11	13	14	GND
PC10	15	16	GND
PB17	17	18	GND
PB16	19	20	GND
PB15	21	22	GND
PB14	23	24	GND
PB13	25	26	GND
PB12	27	28	GND
PB11	29	30	GND
PB10	31	32	GND
PA17	33	34	GND
PA16	35	36	GND
PA15	37	38	GND
PA14	39	40	GND
PA13	41	42	GND
PA12	43	44	GND
PA11	45	46	GND
PA10	47	48	GND
+5 V	49	50	GND

By default the ISA I/O card is set for a base address of 0x300. The author of this project has changed the base address of the card and was set to be 0x200. In order to set the CN1 IDC connector that will be used for writing to the device we send to the address 0x203 (base address + 3) the binary number 10000000.

When we write a byte to the device we are setting the pins PA00 – PA07 to logical 1 or 0, depending to the byte we want to write. For example, if we write to the card the binary number 11110001 the pins PA07, PA06, PA05, PA04, and PA00 will be set to logical 1. The pins PA03, PA02, PA01, will be set to logical 0. Sending to the base address 0x200 a binary number after we set which port range will be used for output it will be written to the pins PA00 – PA07.

At this point we must mention that when we write a value to the device that value stays to the device until we overwrite it with a new value or if we reset the card (reboot the computer).

In order to read from the device we use the CN2 IDC connector and we must set the port range that we will use for that operation. The base address 0x207 (base address + 7) will be used for read (input) from the device. Sending the binary value 10010000 to the base address 0x207 will be set to read.

When we read from the device we are reading the value that exist on the pins PA10 – PA17. Reading the 0x204 address (base address + 4) will return the input from the device. [Reference b7, b8]

Step through the source code of the Module:

In this section of the report we will step through and explain some basic points of the module's source code.

The source code of the module is included in the appendix 2.

In order for the user-space to be able to access the ISA I/O card which is a character device, it must use the file operations. The file operations used in this module are Open, Release, Write and Read.

At the top of the module's source code we can see the libraries used and defining some constant variables. Defining the module as __NO_VERSION__ means that the module has no problem to cooperate with any kernel version. Then, we can see the functions that were used and the declaration of the init and exit functions.

The init function is the first function that will run when the module is loaded into the kernel, correspondently, the exit function is the function that will run when we unload the module from the kernel. These functions "init, exit" will register and un-register the device using the corresponding functions register_device and unregister_device.

The "set operation mode" function will write a specific byte to specific ports of the card in order to set the input and output ports of the card. Those ports are called the registers.

Finally, the most important part of the module is the actions that are being performed by the file operations. In the first case we have the open_card file operation which checks if the device is in use, in order to print a message to the kernel if it is, and then increases the usage count. If the function was executed successfully it will send a second message to the kernel. In order to see these messages we must type the dmesg command which provide us with all the kernel messages.

For the release file operation which in our case we named exit_card, the only thing we must do is to decrease the usage count. Failure to do that will have as result to set us unable to unload the module from the kernel. In order to unload the module we would have to restart the computer.

The write function, using the base address that was set for output will send to the device a value. That value is the incoming value from the user space program. In order for the module to read the value that is entered in the user space program we save that value to a variable. The module will read this value using the pointer of the value and finally output it to the device.

The read function, is trying to get the data from the device reading the specific base address that is set for read. It then passes it to a pointer which is returned to the user space.
[Reference b1, b3, b5, b6]

Writing a script for easier manipulation:

During the research for the first sub-report the author learned how to write simple script files for Linux. That came to be a very helpful when the time of compiling the modules the student wrote a script file for easier manipulation.

```
#!/bin/bash
gcc -D __KERNEL__ -02 -Wall -I"/usr/src/linux/include/" -c isa731.c -o isa731.o
```

```
#!/bin/bash
gcc -Wall user.c -o user.o
```

If we copy the script files into the /sbin directory we can run them without typing the path. That means that will make the operation system to execute the script files as another system command. (e.g. ifconfig)

[Reference b5]

Interrupts:

Interrupts are system calls that are being send to the CPU to let the processor know when something has happened. The hardware (a device) sends a signal to the processor in order to requests for the processor's attention. Generating and manipulating interrupts is a very complex procedure. For that reason interrupts weren't used for the development of the module but this small paragraph was included in order to explain in general how interrupts work.

When a hardware interrupt is generated by a device the CPU sends an IRQ signal to the operating system's kernel. The kernel then according to its configuration can ignore the interrupt and continue its normal schedule of operations (e.g. Round Robin[26]) or check its IRQ table to see if any of the kernel modules has been registered with that interrupt value. If that is the case, it will then probe the module in order to inform it about the situation. It's up to the module then to take further action if needed.

The ISA I/O card provides some jumpers that will allow us to set the interrupts. The first two jumpers (JP5, JP6) will help us to register what kind of IRQ levels the device will use. Jumpers JP1 and JP2 will identify the type of the interrupt triggering. There are two types of interrupt triggering, the rising edge and the falling edge. In the rising edge the interrupt of the device is set to send the logical 0 and when an interrupt occurs it rises to the logical 1. In the falling edge when an interrupt occurs the signal falls from the logical 1 to the logical 0. Using the jumpers JP3, JP4 we can enable,

[26] http://choices.cs.uiuc.edu/~f-kon/RoundRobin/node1.html

disable or use the S/W setting of the interrupts. The S/W option is there to allow us to control the interrupts by other external interrupts.
[Reference b1, b5, b8]

References:

[b1] Daniel P. Bovet, Marco Cesati 2000. *Understanding the Linux Kernel. O'Reilly.*

[b2] Bill Ball 1998. *Teach Yourself Linux in 24 Hours: Second Edition. Sams Publishing.*

[b3] Alessandro Rubini & Jonathan Corbet 2001. *Linux Device Drivers: Second Edition. O'Reilly.*

[b4] Matt Welsh, Matthias Kalle Dalheimer, Lar Kaufman 1999. *Running Linux. O'Reilly.*

[b5] Richard Stones & Neil Matthew 2001. *Beginning Linux Programming.Wrox.*

[b6] Jesse Liberty, David Hovarth 2000. *Teach Yourself C++ for Linux in 21 Days. Sams.*

[b7] Don Anderson, Tom Shanley 1995. *ISA System Architecture. Addison Wesley*

[b8] Advantech. User's Manual for the ISA card PCL-731. 48-bit Digital I/O card.

Appendix 1:

//user space program:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <linux/errno.h>

#ifndef TRUE
# define TRUE 1
#endif

#ifndef FALSE
# define FALSE 0
#endif


int read__ (int fd);
int write__ (int fd);



int main(int argc, char *argv[])
{
   char *dev;
   char *device_name;
   char ar;
   int fd;
   int rcode,parameter,rtr;
   while (TRUE)
     {

                printf("Enter  device  driver  name  or  press  [d]  for
default (/dev/isa731/): ");
                scanf("%s",dev);

                if (*dev=='d')
                    {
                       device_name="/dev/isa731";
                    }
                else
                  {
                     device_name=dev;
                  }

                //file descriptor
                fd  = open (device_name , O_RDWR);

                //if device doesn't exist print an error message
                if (fd == -1)
                  {
                      printf  ("Error:  cannot  open  device  or  device
does not exist\n ...maybe a wrong device name!\n");
                      return -EBUSY;
                   }

        rtr=getc(stdin);

                 parameter=FALSE;
                 while (!(parameter))
                   {
                      printf ("R or W?");
                      scanf ("%c",&ar);

                      if ((ar == 'r') || (ar == 'R'))
                        {
                     rcode = read__(fd);
                      parameter=TRUE;
                        }
                      else if ((ar == 'w') || (ar == 'W'))
                        {
                     rcode = write__(fd);
                     parameter=TRUE;
                        }
                      else
                        {
                     printf("Wrong parameter!\n");
                     parameter=FALSE;
                        }
                   }
                 rtr=close (fd);
     }
                 return rtr;
}//end of main



   int read__ (int fd)//read a byte from the opened file
     {
                 unsigned char byte;

                 read(fd,&byte,1);
                 printf ("Readen : %d\n",byte);
                 return 0;
     }

   int write__  (int  fd)//write  a  byte,that  the  user  enters,  to  the
opened file
     {
                 unsigned char byte;
                 int panos;
                 printf ("Enter a byte to write to device: ");
                 scanf ("%d",&panos);
                 printf ("You entered %d \n",panos);

                 byte = panos;
                 // byte = 0xFF;
                 // byte=strtol(argv[1],stop,16);
                 write(fd,&byte,1);
                 return 0;
     }
```

Appendix 2:

//Module Source Code

```
#ifndef __KERNEL__
#  define __KERNEL__
#endif

#ifndef MODULE
# define MODULE
#endif

#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/errno.h> //less /usr/src/linux/include/asm/errno.h
#include <asm/uaccess.h>
#include <linux/param.h>
#include <linux/ioport.h>
#include <asm/io.h>

#define __NO_VERSION__

EXPORT_NO_SYMBOLS;
//declare device name
char device_name[] = "isa731";
//char *buff;
int MajorNum = 42;
int base_port = 0x200;
int port_range = 8;
int mod_byte1 = 0x90;
int mod_byte2 = 0x80;

//dymanic name declaration @ module load time
MODULE_PARM (device_name,"s");
MODULE_PARM (MajorNum,"i");
MODULE_PARM (base_port,"i");
MODULE_PARM (mod_byte1,"i");
MODULE_PARM (mod_byte2,"i");


//Declaration of functions
static struct file_operations  FileOps;
static int __init start (void);
static void __exit clean (void);
static int register_device (char *device_name, struct file_operations
*FileOps);
static int unregister_device (char *device_name);
static int open_card (struct inode *inode, struct file *file);
static int close_card (struct inode *inode, struct file *file);
static  int  write_card  (struct  file  *filp,  const  char  *ptr,  size_t
count, loff_t *f_pos  );
static  int  read_card  (struct  file  *filp,  char  *ptr,  size_t  count,
loff_t *f_pos);
```

```
static  int  allocate_ports  (unsigned  int  base_port,  unsigned  int
port_range);
static void set_operating_mode();

module_init (start);
module_exit (clean);



// File Operations structure  * * * * * * * * * * * * * * * *
static struct file_operations  FileOps =
{
   .release = close_card,
     .open = open_card,
     .read = read_card,
     .write = write_card,
};

// My INIT function * * * * * * * * * * * * * * * * * * * * *
static int __init start (void)
{
   int check;
   //       int address = base_port + 4;
   //       const char *testbuff=0x00;
   //       int address1 = base_port;   // * * * * * * *
   //       char *ptr;

   check = register_device (device_name, &FileOps);

   if (check<0)
     {
              printk ("<1>Can't register Device :( \n");
              return check;
     }


   check=allocate_ports(base_port,port_range);


   if (check<0)
     {

              printk ("<1>Can't load Module :( \n");
         }

   else
     {
              printk ("<1>Module loaded succesfully :) \n");
     }

   set_operating_mode();

   return check;
}

//My EXIT function * * * * * * * * * * * * * * * * * * * * *
static void __exit clean (void)
{

   unregister_device(device_name);
   release_region(base_port,port_range); //release ports

}


//Set Orerating Mode for ISA I/O card
//PortA0 will be an OUTPUT
//PortA1 will be an INPUT
static void set_operating_mode()
{

   int register_address1 = base_port + 3;
   int register_address2 = base_port + 7;

   outb_p(mod_byte2, register_address1);
   outb_p(mod_byte1, register_address2);

   printk ("<1>SETOPMOD \n");

}



//Allocating Ports to the Registered Device * * * * * * * * * * * * *
* * *
static  int  allocate_ports  (unsigned  int  base_port,  unsigned  int
port_range)
{

  int err;

   if ((err=check_region (base_port,port_range)) < 0)
     return err; // device busy

   request_region (base_port,port_range,device_name);

   return 0;

}



// REGISTER DEVICE * * * * * * * * * * * * * * * * * * * * * *
static int register_device (char *device_name, struct file_operations
*FileOps)
{
   int check;


   if (device_name == NULL)
     device_name = "isa731";

   //register device with kernel and return >=0 for true || <=0 for
false
   //use the current Major Number for the specific device according
the File Operations
   check = register_chrdev(MajorNum, device_name, FileOps);

   if (check<0)
     {
              printk ("<1>Can't register device :( \n");
                return check;
     }
   return 0;
}

// UNREGISTER DEVICE * * * * * * * * * * * * * * * * * * * * *
static int unregister_device (char *device_name)
{

   int check;
   check = unregister_chrdev(MajorNum, device_name);

   if (check<0)
     {
                printk  ("<1>Can't  unregister  device....  Maybe  in
use...\n");
                return check;
     }
   return 0;
}


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * *
// FILE OPERATIONS
// OPEN
// READ
// WRITE
// RELEASE
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * *
//
//
//
static int open_card (struct inode *inode, struct file *file)
{
   printk ("<1>OPEN \n");
   if(MOD_IN_USE>0)
     return -EBUSY;  //if the card is beeing accesed already return
"Device or Resource Busy"
   //increment usage count
   MOD_INC_USE_COUNT;
   printk ("<1>OPEN2 \n");
   return 0;
}


static int close_card (struct inode *inode, struct file *file)
{

   //decrement usage count
   MOD_DEC_USE_COUNT;
   printk ("<1>CLOSE");
   return 0;
}


static  int  write_card  (struct  file  *filp,  const  char  *ptr,  size_t
count, loff_t *f_pos  )
{
   int address = base_port;
   int buff = *ptr;

   outb(buff, address );   //write to device

   return 0;

}


static int read_card (struct file *filp, char *ptr, size_t count,
loff_t *f_pos)
{
   unsigned char address =  base_port + 4;   // * * * * * * *
   int n = count;
   while (n--)
     {
        printk ("<1>READ");
        *(ptr++) = inb(address);  // get data from device
        printk ("<1>READEN %d from device",*ptr);
     }
   return (int)count;
}
```

Progress Reports:

Linux Device Drivers

Progress Report of Level 3 Project
1st – 8th November 2002

Review Tasks Set Last Week

- This is the first week.

Tasks Set This Week

- Preparation for the project presentation.

Forward look to tasks for next stage

- Familiarize with Red Hat Linux 8.0
- Research project requirements

Linux Device Drivers

Progress Report of Level 3 Project
8th – 15th November 2002

Review Tasks Set Last Week

- A new version of the work plan has been created.

Tasks Set This Week

- Familiarization with Red Had Linux environment is in progress.
- Research about module programming is started.
- Researching how PCI bus interface works.
- A PCI I/O card has been proposed to the supervisor via e-mail in order to decide if it's appropriate for the project.

Forward look to tasks for next stage

- Continuing the research about Red Hat Linux environment, driver programming, PCI bus interface and PCI I/O card specifications.

Linux Device Drivers

Progress Report of Level 3 Project
15th – 22nd November 2002

Review Tasks Set Last Week

- Familiarization with Red Had Linux environment is in progress.

Tasks Set This Week

- Familiarization with Red Had Linux environment is in progress.
- Research about module programming is in progress.
- Researching how PCI bus interface works.
- Writing some simple shell scripts to familiarize how the shells work.

Forward look to tasks for next stage

- Continuing the research about Red Hat Linux environment, driver programming.

- Understanding PCI I/O card specifications.

Linux Device Drivers

Progress Report of Level 3 Project
22nd – 29th November 2002

Review Tasks Set Last Week

- Research on "How Linux Works" is in progress.
- Research on "How to write a Kernel Driver" is in progress

Tasks Set This Week

- Familiarization with Red Had Linux environment is in progress.
- Research required C++ source code for Linux.
- Writing some simple shell scripts to familiarize how the shells work.
- Trying to write some very simple C++ programs in console environment using the Jed editor.

Forward look to tasks for next stage

- Red Hat Linux has a customized Kernel and patched. In order to avoid future problem (about Kernel versions) I would like to use the official Kernel. That's why I choused slackware 8.1 which uses the 2.4.18 official Kernel.

Linux Device Drivers

Progress Report of Level 3 Project
29th – 6th November 2002

Review Tasks Set Last Week

- Research on "How Linux Works" is in progress.
- Research on "How to write a Kernel Driver" is in progress

Tasks Set This Week

- Familiarization with Red Had Linux environment is in progress.
- Research required C++ source code for Linux.
- Learning how to recompile the Linux Kernel in order to make it work with all hardware devices and be able to see the NTFS partition of windows.
- The I/O card has arrived which has been ordered from the web site www.advantech.com but unfortunately they have send the wrong card. Instead of a PCI I/O card they've send me an ISA I/O card.
- Searching for some information in the Linux Journals.

Forward look to tasks for next stage

- Contacting the supervisor in order to ask what should I do with the ISA card? I've already contact the company and ask them what I should do in order to return the card. Because I 'm a foreign student and I won't be in the UK for the Christmas period it would be impossible for the company which I ordered the card to contact in case they need any information. From the other hand I cannot wait to return the card after Christmas period because they might not accept it.

Linux Device Drivers

Progress Report of Level 3 Project
<u>6th – 13th December 2002</u>

Review Tasks Set Last Week

- Research on "How Linux Works" is in progress.
- Research on "How to write a Kernel Driver" is in progress

Tasks Set This Week

- Familiarization with Slackware Linux environment is in progress.
- Research required C++ source code for Linux.
- Learning how to recompile the Linux Kernel in order to make it work with all hardware devices and be able to see the NTFS partition of windows.
- Working on the specifications of the ISA I/O card
- Searching for some information in the Linux Journal and Linux magazine.
- Some part of the sub-report that is to be submitted on Friday 10-12-03 has been written.

Forward look to tasks for next stage

- Provide to the supervisor a sample of the work that has been done until now and the part of the sub-report that has been implemented.

Linux Device Drivers

Progress Report of Level 3 Project
<u>13th – 10th January 2003</u>

Review Tasks Set Last Week

- Research on "How Linux Works" is in progress.

Tasks Set This Week

- Start writing the sub-report.

Forward look to tasks for next stage

- Start the second part of the project.

Linux Device Drivers

Progress Report of Level 3 Project
10th – 17th January 2003

Review Tasks Set Last Week

- Working on Linux architecture.

Tasks Set This Week

- Trying to understand important parts of the Kernel.
- Started writing some C code for the kernel.
- Understanding how read and write functions works along with the init and exit.

Forward look to tasks for next stage

- Working on major numbers and adding devices into the /dev directory.

Linux Device Drivers

Progress Report of Level 3 Project
17th – 24th January 2003

Review Tasks Set Last Week

- Review the work that has been done on kernel programming and trying to write some source code.

Tasks Set This Week

- A chunk of source code had been produced in order to develop a simple module. This module is using the file operations and is attempting to read and write in a buffer through the kernel.
- The module has been compiled in order to make it a module.o file, using the following command: gcc –Wall -O2 –c main.c –o greg.o
- A device (file) has been added into the /dev directory using the following command: mknod /dev/greg c 42 1
- Using the following command we are trying to link a module to the kernel. insmod greg.o
- The module has been compiled successfully after many attempts but with a few warnings. Unfortunately even when the module was linked to the kernel it didn't work as it should.
- Another problem was that the module could not be unloaded from the kernel so we had to reboot the system in order to try again.
- The module that was produce is attached to the minutes of this meeting.

Forward look to tasks for next stage

- Working on the source code in order to understand the kernel in a better level.
- Further research on kernel programming.

Example of source code:

```
#ifndef __KERNEL__
#  define __KERNEL__
#endif

#ifndef MODULE
#  define MODULE
#endif
```

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/errno.h>
#include <asm/uaccess.h>
#include <linux/param.h>


#define __NO_VERSION__

EXPORT_NO_SYMBOLS;
//declare device name
char device_name[] = "greg";
char *buff;
int MajorNum = 42;

//dymanic name declaration @ module load time
MODULE_PARM (device_name,"s");
MODULE_PARM (MajorNum,"i");

// Declaration of functions
static struct file_operations  FileOps;
static int __init start (void);
static void __exit clean (void);
static int register_device (char *device_name, struct file_operations
*FileOps);
static int unregister_device (char *device_name);
static int open (struct inode *inode, struct file *file);
static int close (struct inode *inode, struct file *file);
static int write (struct file *filp, const char *ptr, size_t count,
loff_t *f_pos  );
static int read (struct file *filp, char *ptr, size_t count, loff_t
*f_pos);

module_init (start);
module_exit (clean);

// File Operations structure  * * * * * * * * * * * * * * * * *
static struct file_operations  FileOps =
{
   .release = close,
   .open = open,
   .read = read,
   .write = write,
};

// My INIT function * * * * * * * * * * * * * * * * * * * * * *
static int __init start (void)
{
       int check;
       check = register_device (device_name, &FileOps);

       if (check<0)
         {
            printk ("<1>Can't load Module :( \n");
         }
       else
         {
            printk ("<1>Module loaded succesfully :) \n");
          }
       return check;
}

//My EXIT function * * * * * * * * * * * * * * * * * * * * * *
static void __exit clean (void)
{
          unregister_device(device_name);
}

// REGISTER DEVICE * * * * * * * * * * * * * * * * * * * * * *
static int register_device (char *device_name, struct file_operations
*FileOps)
{
       int check;
       if (device_name == NULL)
             device_name = "greg";

       //register device with kernel and return >=0 for true || <=0
for false
       //use the current Major Number for the specific device
according the File
       //Operations
       check = register_chrdev(MajorNum, device_name, FileOps);

       if (check<0)
         {
          printk ("<1>Can't register device :( \n");
          return check;
         }
       return 0;
}

// UNREGISTER DEVICE * * * * * * * * * * * * * * * * * * * * * *
static int unregister_device (char *device_name)
{
       int check;
       check = unregister_chrdev(MajorNum, device_name);

       if (check<0)
         {
         printk ("<1>Can't unregister device.... Maybe in use...\n");
         return check;
         }
return 0;
}

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * *
// FILE OPERATIONS
// OPEN - READ - WRITE - RELEASE
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * *

static int open (struct inode *inode, struct file *file)
{
       //increment usage count
       MOD_INC_USE_COUNT;
       return 0;
}

static int close (struct inode *inode, struct file *file)
```

```
{
       //decrement usage count
       MOD_DEC_USE_COUNT;
       return 0;
}

static int write (struct file *filp, const char *ptr, size_t count,
loff_t *f_pos  )
{
       int inp;
/*       if (MOD_IN_USE>1)
         {
          printk ("<1>Can't write to the device! \n");
          return -EBUSY;
       }
 */
       if (copy_from_user(buff, ptr, count))
                     return -EFAULT;
       inp = (int) buff;
       printk ("<1>The input was %d",inp);
       return 0;
}
static int read (struct file *filp, char *ptr, size_t count, loff_t
*f_pos)
{
   if (copy_to_user(ptr, buff, count))
                     return -EFAULT;

   return 0;

}
```

Linux Device Drivers

Progress Report of Level 3 Project
24th – 31st January 2003

Review Tasks Set Last Week

- Review the work that has been done on kernel programming and trying to write some more source code.

Tasks Set This Week

- Working on the source code that has been submitted to the supervisor last week for review and feedback.

Forward look to tasks for next stage

- Further research on kernel programming.

Linux Device Drivers

Progress Report of Level 3 Project
<u>31st – 7th February 2003</u>

Review Tasks Set Last Week

- Review the work that has been done on kernel programming and trying to write some more source code.

Tasks Set This Week

- Working on the source code that has been submitted in the previous weeks for review and feedback

Forward look to tasks for next stage

- Working on the structure of the module, trying to identify some key points and finally to compile a simple module.

Linux Device Drivers

Progress Report of Level 3 Project
7th – 14th February 2003

Review Tasks Set Last Week

- Review the work that has been done on kernel programming and trying to write some more source code.

Tasks Set This Week

- Working on the source code that has been submitted in the previous weeks for review and feedback

Forward look to tasks for next stage

- Trying to develop and use the file operations (write and read).

Linux Device Drivers

Progress Report of Level 3 Project
<u>14th – 21st February 2003</u>

Review Tasks Set Last Week

- Review the work that has been done on kernel programming.
- Writing source code for the module.

Tasks Set This Week

- Working on the source code that has been submitted in the previous weeks for review and feedback
- An example of the new version of the source code will be submitted to the supervisor on the next meeting.
- The following command was used in order to compile the new version of the module into an executable script file.

```
#!/bin/bash
gcc -D __KERNEL__ -Wall -I"/usr/src/linux/include/" -c
labcard.c -o labcard.o
```

Forward look to tasks for next stage

- Trying to develop and use the file operations (write and read).

Updated Source Code:

<u>The libraries:</u>

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/errno.h> //less /usr/src/linux/include/asm/errno.h
#include <asm/uaccess.h>
#include <linux/param.h>
#include <linux/ioport.h>
#include <asm/io.h>
```

<u>The declaration of variables:</u>

```
char device_name[] = "greg";
char *buff;
int MajorNum = 42;
int base_port = 0x300;
int port_range = 8;
int mod_byte1 = 0x90;
int mod_byte2 = 0x80;
```

INIT & EXIT function:

```
// My INIT function * * * * * * * * * * * * * * * * * * * * *
static int __init start (void)
{
       int check;

       check = register_device (device_name, &FileOps);

       if (check<0)
         {
            printk ("<1>Can't register Device :( \n");
          return check;
         }


       check=allocate_ports(base_port,port_range);


       if (check<0)
         {

            printk ("<1>Can't load Module :( \n");
         }

       else
         {
          printk ("<1>Module loaded succesfully :) \n");
         }

       set_operating_mode();
       printk ("<1>Module loaded succesfully :| INIT \n");
       return check;
}

//My EXIT function * * * * * * * * * * * * * * * * * * * * *
static void __exit clean (void)
{

       unregister_device(device_name);

       release_region(base_port,port_range); //release ports

}
```

Set Operation Mode function:

```
//Set Orerating Mode for ISA I/O card
//PortA will be an INPUT
//PortB will be an OUTPUT
static void set_operating_mode()
{

   int register_address1 = base_port + 3;
   int register_address2 = base_port + 7;

   outb_p(register_address1,mod_byte1);
   outb_p(register_address2,mod_byte2);
   printk ("<1>Module loaded succesfully :| SETOPMOD \n");

}
```

Register & Unregister Device

```
// REGISTER DEVICE * * * * * * * * * * * * * * * * * * * * * * *
static int register_device (char *device_name, struct file_operations
*FileOps)
{
       int check;


       if (device_name == NULL)
             device_name = "greg";

       //register device with kernel and return >=0 for true || <=0
for false
       //use the current Major Number for the specific device
according the File Operations
       check = register_chrdev(MajorNum, device_name, FileOps);

       if (check<0)
         {
          printk ("<1>Can't register device :( \n");
          return check;
         }
       return 0;
}

// UNREGISTER DEVICE * * * * * * * * * * * * * * * * * * * * * *
static int unregister_device (char *device_name)
{

       int check;
       check = unregister_chrdev(MajorNum, device_name);

       if (check<0)
         {
         printk ("<1>Can't unregister device.... Maybe in use...\n");
         return check;
         }
return 0;
}
```

File Operation (Open & Release)

```
static int open_card (struct inode *inode, struct file *file)
{
       printk ("<1>Module loaded succesfully :| OPEN \n");
       if(MOD_IN_USE>0)
       return -EBUSY;  //if the card is beeing accesed already return
"Device or Resource Busy"
       //increment usage count
       MOD_INC_USE_COUNT;
    printk ("<1>Module loaded succesfully :| OPEN2 \n");
       return 0;
}


static int close_card (struct inode *inode, struct file *file)
{
        //decrement usage count
        MOD_DEC_USE_COUNT;
   printk ("<1>CLOSE");
   return 0;
}
```

Adding a device into the /dev directory:

```
mknod /dev/greg c 0 42
```

Linux Device Drivers

Progress Report of Level 3 Project
21st – 28th February 2003

Review Tasks Set Last Week

- Writing source code for the module.

Tasks Set This Week

- Working on the source code that has been submitted the previous week to the supervisor in order to implement the read and write file operations.
- The stage of the code will be submitted to the supervisor on the next meeting.
- Even though the file operations read and write are in the implementation stage for about 3 weeks, still we are not able to read from the ISA card successfully.
- The student started writing a user space application.

Forward look to tasks for next stage

- Trying to implement the file operations (write and read).

The following is the source code of the ISA I/O card module that has implemented up to date.

```
#ifndef __KERNEL__
#  define __KERNEL__
#endif

#ifndef MODULE
#  define MODULE
#endif

#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/errno.h> //less /usr/src/linux/include/asm/errno.h
#include <asm/uaccess.h>
#include <linux/param.h>
#include <linux/ioport.h>
```

```
#include <asm/io.h>

#define __NO_VERSION__

EXPORT_NO_SYMBOLS;
//declare device name
char device_name[] = "greg";
char *buff;
int MajorNum = 42;
int base_port = 0x300;
int port_range = 8;
int mod_byte1 = 0x90;
int mod_byte2 = 0x80;

//dymanic name declaration @ module load time
MODULE_PARM (device_name,"s");
MODULE_PARM (MajorNum,"i");
MODULE_PARM (base_port,"i");
MODULE_PARM (mod_byte1,"i");
MODULE_PARM (mod_byte2,"i");


//Declaration of functions
static struct file_operations  FileOps;
static int __init start (void);
static void __exit clean (void);
static int register_device (char *device_name, struct file_operations
*FileOps);
static int unregister_device (char *device_name);
static int open_card (struct inode *inode, struct file *file);
static int close_card (struct inode *inode, struct file *file);
static int write_card (struct file *filp, const char *ptr, size_t
count, loff_t *f_pos  );
static int read_card (struct file *filp, char *ptr, size_t count,
loff_t *f_pos);
static int allocate_ports (unsigned int base_port, unsigned int
port_range);
static void set_operating_mode();

module_init (start);
module_exit (clean);



// File Operations structure  * * * * * * * * * * * * * * * * *
static struct file_operations  FileOps =
{
   .release = close_card,
   .open = open_card,
   .read = read_card,
   .write = write_card,
};

// My INIT function * * * * * * * * * * * * * * * * * * * * * *
static int __init start (void)
{
       int check;

       check = register_device (device_name, &FileOps);

       if (check<0)
```

```
          {
             printk ("<1>Can't register Device :( \n");
           return check;
          }


       check=allocate_ports(base_port,port_range);


       if (check<0)
          {

             printk ("<1>Can't load Module :( \n");
          }

       else
          {
           printk ("<1>Module loaded succesfully :) \n");
          }

       set_operating_mode();
       printk ("<1>Module loaded succesfully :| INIT \n");
       return check;
}

//My EXIT function * * * * * * * * * * * * * * * * * * * * * *
static void __exit clean (void)
{

       unregister_device(device_name);

       release_region(base_port,port_range); //release ports

}


//Set Orerating Mode for ISA I/O card
//PortA will be an INPUT
//PortB will be an OUTPUT
static void set_operating_mode()
{

   int register_address1 = base_port + 3;
   int register_address2 = base_port + 7;

   outb_p(register_address1,mod_byte1);
   outb_p(register_address2,mod_byte2);

   printk ("<1>Module loaded succesfully :| SETOPMOD \n");

}



//Allocating Ports to the Registered Device * * * * * * * * * * * * *
* * *
static int allocate_ports (unsigned int base_port, unsigned int
port_range)
{

       int err;
```

```
	if ((err=check_region (base_port,port_range)) < 0)
		return err; // device busy

	request_region (base_port,port_range,device_name);

	return 0;

}



// REGISTER DEVICE * * * * * * * * * * * * * * * * * * * * *
static int register_device (char *device_name, struct file_operations
*FileOps)
{
	int check;


	if (device_name == NULL)
		device_name = "greg";

	//register device with kernel and return >=0 for true || <=0
for false
	//use the current Major Number for the specific device
according the File Operations
	check = register_chrdev(MajorNum, device_name, FileOps);

	if (check<0)
	  {
	   printk ("<1>Can't register device :( \n");
	   return check;
	  }
	return 0;
}

// UNREGISTER DEVICE * * * * * * * * * * * * * * * * * * * * *
static int unregister_device (char *device_name)
{

	int check;
	check = unregister_chrdev(MajorNum, device_name);

	if (check<0)
	  {
	  printk ("<1>Can't unregister device.... Maybe in use...\n");
	  return check;
	  }
return 0;
}


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * *
// FILE OPERATIONS
// OPEN
// READ
// WRITE
// RELEASE
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * *
```

```
//
//

static int open_card (struct inode *inode, struct file *file)
{
       printk ("<1>Module loaded succesfully :| OPEN \n");
       if(MOD_IN_USE>0)
       return -EBUSY;  //if the card is beeing accesed already return
"Device or Resource Busy"
       //increment usage count
       MOD_INC_USE_COUNT;
    printk ("<1>Module loaded succesfully :| OPEN2 \n");
       return 0;
}


static int close_card (struct inode *inode, struct file *file)
{

       //decrement usage count
       MOD_DEC_USE_COUNT;
   printk ("<1>CLOSE");
   return 0;
}


static int write_card (struct file *filp, const char *ptr, size_t
count, loff_t *f_pos  )
{

  //     int address = base_port + 4;   // * * * * * * *
 //  printk ("<1>WRITE");

 //  outb_p(*ptr, address);   //write to device

 //      printk ("<1>Output Data...\n");
 //  printk ("<1>WRITE2");

   int retval = count;
   int address = base_port + 4;

       while (count--)
     {
        outb_p(*(ptr++), address);
     }

   return retval;

}



static int read_card (struct file *filp, char *ptr, size_t count,
loff_t *f_pos)
{

   int address = base_port;   // * * * * * * *
    int n = count;
   printk ("<1>READ");
  // *ptr=inb_p(address);  // get data from device
```

```
        while (n--)
      {
         *(ptr++) = inb_p(address);  // get data from device
      }



   printk ("<1>READ2");
   return (int) count;

}
```

Linux Device Drivers

Progress Report of Level 3 Project
28th – 7th March 2003

Review Tasks Set Last Week

- Writing source code for the module.

Tasks Set This Week

- Working on the source code that has been submitted the previous week to the supervisor in order to implement the read and write file operations.
- The stage of the code will be submitted to the supervisor on the next meeting.
- Finally the student achieved to make the read function to work.
- The student made progress on the source code that is trying to write about the user space program.
- The write function sends a number successfully to the card but we cannot see that value using the read function. Probably a wrong connection has been made to the cable of the card or we are using the wrong base address.

Forward look to tasks for next stage

- Trying to find out why we can't read the value that we are sending to the ISA I/O card using the write function. Using the dmesg command in order to see the outputs of the kernel we can see that the module had successfully send a value to the card.

The following is the source code of the ISA I/O card module that has implemented up to date.

```
#ifndef __KERNEL__
#  define __KERNEL__
#endif

#ifndef MODULE
#  define MODULE
#endif

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

```
#include <linux/fs.h>
#include <linux/errno.h> //less /usr/src/linux/include/asm/errno.h
#include <asm/uaccess.h>
#include <linux/param.h>
#include <linux/ioport.h>
#include <asm/io.h>

#define __NO_VERSION__

EXPORT_NO_SYMBOLS;
//declare device name
char device_name[] = "greg";
char *buff;
int MajorNum = 42;
int base_port = 0x300;
int port_range = 8;
int mod_byte1 = 0x90;
int mod_byte2 = 0x80;

//dymanic name declaration @ module load time
MODULE_PARM (device_name,"s");
MODULE_PARM (MajorNum,"i");
MODULE_PARM (base_port,"i");
MODULE_PARM (mod_byte1,"i");
MODULE_PARM (mod_byte2,"i");


//Declaration of functions
static struct file_operations  FileOps;
static int __init start (void);
static void __exit clean (void);
static int register_device (char *device_name, struct file_operations
*FileOps);
static int unregister_device (char *device_name);
static int open_card (struct inode *inode, struct file *file);
static int close_card (struct inode *inode, struct file *file);
static int write_card (struct file *filp, const char *ptr, size_t
count, loff_t *f_pos  );
static int read_card (struct file *filp, char *ptr, size_t count,
loff_t *f_pos);
static int allocate_ports (unsigned int base_port, unsigned int
port_range);
static void set_operating_mode();

module_init (start);
module_exit (clean);



// File Operations structure  * * * * * * * * * * * * * * * *
static struct file_operations  FileOps =
{
   .release = close_card,
   .open = open_card,
   .read = read_card,
   .write = write_card,
};

// My INIT function * * * * * * * * * * * * * * * * * * * * *
static int __init start (void)
{
       int check;
       int address = base_port + 4;
       const char *testbuff=0x00;
       int address1 = base_port;   // * * * * * * * *
       char *ptr;

       check = register_device (device_name, &FileOps);

       if (check<0)
         {
            printk ("<1>Can't register Device :( \n");
          return check;
         }


       check=allocate_ports(base_port,port_range);


       if (check<0)
         {

            printk ("<1>Can't load Module :( \n");
         }

       else
         {
          printk ("<1>Module loaded succesfully :) \n");
         }

       set_operating_mode();

       return check;
}

//My EXIT function * * * * * * * * * * * * * * * * * * * * *
static void __exit clean (void)
{

       unregister_device(device_name);

       release_region(base_port,port_range); //release ports

}


//Set Orerating Mode for ISA I/O card
//PortA will be an INPUT
//PortB will be an OUTPUT
static void set_operating_mode()
{

   int register_address1 = base_port + 3;
   int register_address2 = base_port + 7;

   outb_p(register_address1,mod_byte1);
   outb_p(register_address2,mod_byte2);

   printk ("<1>Module loaded succesfully :| SETOPMOD \n");

}
```

```
//Allocating Ports to the Registered Device * * * * * * * * * * * * *
* * *
static int allocate_ports (unsigned int base_port, unsigned int
port_range)
{

  int err;

  if ((err=check_region (base_port,port_range)) < 0)
       return err; // device busy

   request_region (base_port,port_range,device_name);

   return 0;

}



// REGISTER DEVICE * * * * * * * * * * * * * * * * * * * * * *
static int register_device (char *device_name, struct file_operations
*FileOps)
{
   int check;


   if (device_name == NULL)
     device_name = "greg";

   //register device with kernel and return >=0 for true || <=0 for
false
   //use the current Major Number for the specific device according
the File Operations
   check = register_chrdev(MajorNum, device_name, FileOps);

   if (check<0)
         {
          printk ("<1>Can't register device :( \n");
          return check;
         }
   return 0;
}

// UNREGISTER DEVICE * * * * * * * * * * * * * * * * * * * * * *
static int unregister_device (char *device_name)
{

   int check;
   check = unregister_chrdev(MajorNum, device_name);

       if (check<0)
     {
      printk ("<1>Can't unregister device.... Maybe in use...\n");
      return check;
     }
   return 0;
}
```

```
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * *
// FILE OPERATIONS
// OPEN
// READ
// WRITE
// RELEASE
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * *
//
//

static int open_card (struct inode *inode, struct file *file)
{
   printk ("<1>Module loaded succesfully :| OPEN \n");
       if(MOD_IN_USE>0)
     return -EBUSY;  //if the card is beeing accesed already return
"Device or Resource Busy"
   //increment usage count
   MOD_INC_USE_COUNT;
   printk ("<1>Module loaded succesfully :| OPEN2 \n");
   return 0;
}


static int close_card (struct inode *inode, struct file *file)
{

   //decrement usage count
   MOD_DEC_USE_COUNT;
   printk ("<1>CLOSE");
   return 0;
}


static int write_card (struct file *filp, const char *ptr, size_t
count, loff_t *f_pos  )
{
   int address = base_port + 4;
   int buff =  *ptr;

   printk ("<1>WRITE %d to device\n\n\n",buff);
   outb_p(buff, address);   //write to device

   return 0;

}

static int read_card (struct file *filp, char *ptr, size_t count,
loff_t *f_pos)
{
   int address = base_port;   // * * * * * * *

   printk ("<1>READ");
   *ptr=inb_p(address);  // get data from device
   printk ("<1>READ2");
   return 0;


}
```

The following is the source code of the User-Space program that has implemented up to date.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <linux/errno.h>

#ifndef TRUE
# define TRUE 1
#  endif

#ifndef FALSE
#   define FALSE 0
#  endif


int read__ (int fd);
int write__ (int fd);



int main(int argc, char *argv[])
{
   //   char dev[11];
   char *device_name;
   char ar;
   int fd;
   int rcode;
   //   while (TRUE)
   //     {

   /* printf("Enter device driver name or press [d] for default
(/dev/greg/): ");
 scanf("%c",dev);
 *
 if (dev=="d")
 {
 device_name="/dev/greg/";
        }
 else
 {*/
   //      device_name=dev;
//      }

   device_name="/dev/greg";

   fd  = open (device_name , O_RDWR);

   //if device doesn't exist print an error message
   if (fd == -1)
     {

          printf ("Error: cannot open device or device does not
exist\n ...maybe a wrong device name!\n");
```

```
          return -EBUSY;
        }
   printf ("R or W?");
   scanf ("%c",&ar);

   if ((ar == 'r') || (ar == 'R'))
     {
      rcode = read__(fd);
     }
      else if ((ar == 'w') || (ar == 'W'))
     {
      rcode = write__(fd);
     }
   else
     {
      printf("Wrong parameter!\nUsage : user r\t read from device\n
\t\t user -w\t write to device\n");
     }
   //     }
   return 0;
}//end of main



   int read__ (int fd)
     {
      unsigned char byte;

      read(fd,&byte,1);
      printf ("Readen : %d\n",byte);
      return 0;
     }

   int write__ (int fd)
     {
      unsigned char byte;
//    printf ("Enter a byte to write to device: ");
//    scanf ("%s",&byte);
//    printf ("You entered %c \n",byte);

      byte = 0x3;

      write(fd,&byte,1);
      return 0;
     }
```

Linux Device Drivers

Progress Report of Level 3 Project
<u>7th – 14th March 2003</u>

Review Tasks Set Last Week

- Writing source code for the module.
- Started writing the second sub-report.

Tasks Set This Week

- Working on the source code that has been submitted the previous week to the supervisor in order to implement the write file operation.
- The deliverables of the second stage of the project will be discussed with the supervisor.
- The student is trying to combine all the research and the work that has been done up to date in order to write the second sub-report that is to be submitted on week 22.

Forward look to tasks for next stage

- Trying to make the write file operation to work correctly.
- Continue on writing the second sub-report of the project.

Linux Device Drivers

Progress Report of Level 3 Project
14th – 21st March 2003

Review Tasks Set Last Week

- Testing the source of the module.
- Continue writing the second sub-report.

Tasks Set This Week

- Working on the source code that has been submitted the previous week to the supervisor in order to test and debug the program.
- The main structure of the module has finished and some testing has been done also.
- The writing of the second sub-report is in progress.
- The student in order to test the card has bought some LEDs a 50 pin IDC cable (and some extra electronic material). Connecting the 50 pin cable to the card and having 8 leds connected at the end of the cable, the student tried to test if we he is sending the commands to the correct base address of the card. In order to do that more easily the student boot the PC with MS-DOS 6.22 and having installed to a second PC the Borland C compiler wrote some programs for the ISA card.
- The base address was changed to 0x200 because it seemed to be a problem with the previous (0x300).
- A demo program that was in the user’s manual of the card was used also.
- The custom made programs of the student and the program from the user’s manual had the same result. The student was able to write to the device and see the leds on, but he couldn’t read from the device for some reason.
- Finally, a last modification has been made to the module and to the user space program and the write operation to the device was successful and the output was able to be seen at the LEDs.
- There is a chance that there is some kind of problem with the card and that is the reason that we cannot read from the device. This will be examined during the testing process in for the third sub-report.
- The source code of the programs that were used to test the card, the latest source code of the module and the user-space program are included below

Forward look to tasks for next stage

- Combine all the work that has been done up to date in order to finish and submit the final version of the second sub-report for next week.

The program from the user's manual of the ISA card that was used in order to test the card.

```
#include<stdio.h>
#include<conio.h>
#include<process.h>
//#include<dos.h>
main()
{
int base = 0x200;
int portA;
int portB;
int portC;
int i,j;

clrscr();
gotoxy(25,3);
textattr(0x70);
cputs("PCL-731 Testing Program");
gotoxy(11,6);
printf("PortA0 output value -> ");
gotoxy(11,8);
printf("PortB0 output value -> ");
gotoxy(11,10);
printf("PortC0 output value -> ");
gotoxy(43,6);
printf("PortA1 readback -> ");
gotoxy(43,8);
printf("PortB1 readback -> ");
gotoxy(43,10);
printf("PortC1 readback -> ");

//outportb (base+3,0x80);
for (j=0;j<0x100;j++)
{
      outportb (base,j);
      gotoxy(34,6);
      printf("%2x",j);
      portA = inportb (base);
      gotoxy(63,6);
      printf("%2x",portA);
      if (portA!=j)
            {
                  printf("\7"); /*beep*/
                  gotoxy(30,13);
                  textattr(0x09);
                  cprintf("PortA1 readback error!");
                  getch();
                  //exit(1);
            }
      outportb (base+1,j);
      gotoxy(34,8);
      printf("%2x",j);
      portB = inportb (base+1);
      gotoxy(63,8);
      printf("%2x",portB);
      if (portB!=j)
            {
                  printf("\7"); /*beep*/
                  gotoxy(30,13);
                  textattr(0x09);
                  cprintf("PortB1 readback error!");
                  getch();
                  //exit(1);
            }
      outportb (base+2,j);
      gotoxy(34,10);
      printf("%2x",j);
      portC = inportb (base+2);
      gotoxy(63,10);
      printf("%2x",portC);
      if (portC!=j)
            {
                  printf("\7"); /*beep*/
                  gotoxy(30,13);
                  textattr(0x09);
                  cprintf("PortC1 readback error!");
                  getch();
                  //exit(1);
            }
} /* end of for */
return(0);
} /* end of main */
```

The source code of the program that is trying to write to the device from MS-DOS

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
/*include <dos.h>*/

int main (int argc, char *argv[])
{

     int port,byte;
     char *stop,*stop1;

     if (argc > 3)
          printf ("\nThis program supports only two options... \n
Type testp -h for help");
     else
         {
               if ((argv[1] == "-h") || (argc==1))
         printf (" This program is used to write a byte to the
specified port\n usage:             testp port_to_write(hex) byte_to
write");
         else
             {
            if (argv[1] == "r")

                       /*{
                       port=strtol(argv[2],stop,16);8/
                  byte = inport(0x110);
                  printf ("VALUE = %d ",byte);
            }*/
            else
            {
                  port=strtol(argv[1],stop,16);
                  byte=strtol(argv[2],stop1,2);
                  printf ("\nport: %x byte: %d ",port,byte);
                  printf ("Ok \n");
                  outport (port,byte);
            }
      }
}
getch();
return 0;
}
```

The source code of the program that is trying to read from the device (MS-DOS)

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

main ()
{
int byte,byte1,byte2,byte3,byte4,byte5;
printf ("Reading....\n");
int base = 0x200;

byte = inportb (base);
byte1 = inportb (base + 1);
byte2 = inportb (base + 2);
byte3 = inportb (base + 4);
byte4 = inportb (base + 5);
byte5 = inportb (base + 6);


printf ("0x200=%d 0x201=%x 0x202=%x 0x204=%d  0x205=%x
0x206=%x",byte,byte1,byte2,byte3,byte4,byte5);

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

The source code of the user space program

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <linux/errno.h>

#ifndef TRUE
# define TRUE 1
#endif
```

```
#ifndef FALSE
# define FALSE 0
#endif


int main(int argc, char *argv[])
{
   char *device_name;
   int fd;
   unsigned char byte=0xA9;
   device_name="/dev/greg";

   fd  = open (device_name , O_RDWR);

   //if device doesn't exist print an error message
   if (fd == -1)
     {

      printf ("Error: cannot open device or device does not exist\n
...maybe a wrong device name!\n");
      return -EBUSY;
     }

   write(fd,&byte,1);
   return 0;
}
```

The source code of the module

```
#ifndef __KERNEL__
#  define __KERNEL__
#endif

#ifndef MODULE
# define MODULE
#endif

#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/errno.h> //less /usr/src/linux/include/asm/errno.h
#include <asm/uaccess.h>
#include <linux/param.h>
#include <linux/ioport.h>
#include <asm/io.h>

#define __NO_VERSION__

EXPORT_NO_SYMBOLS;
//declare device name
char device_name[] = "greg";
//char *buff;
int MajorNum = 42;
int base_port = 0x200;
int port_range = 8;
int mod_byte1 = 0x90;
int mod_byte2 = 0x80;
```

```
//dymanic name declaration @ module load time
MODULE_PARM (device_name,"s");
MODULE_PARM (MajorNum,"i");
MODULE_PARM (base_port,"i");
MODULE_PARM (mod_byte1,"i");
MODULE_PARM (mod_byte2,"i");


//Declaration of functions
static struct file_operations  FileOps;
static int __init start (void);
static void __exit clean (void);
static int register_device (char *device_name, struct file_operations
*FileOps);
static int unregister_device (char *device_name);
static int open_card (struct inode *inode, struct file *file);
static int close_card (struct inode *inode, struct file *file);
static int write_card (struct file *filp, const char *ptr, size_t
count, loff_t *f_pos  );
static int read_card (struct file *filp, char *ptr, size_t count,
loff_t *f_pos);
static int allocate_ports (unsigned int base_port, unsigned int
port_range);
static void set_operating_mode();

module_init (start);
module_exit (clean);



// File Operations structure  * * * * * * * * * * * * * * * *
static struct file_operations  FileOps =
{
   .release = close_card,
     .open = open_card,
     .read = read_card,
     .write = write_card,
};

// My INIT function * * * * * * * * * * * * * * * * * * * * *
static int __init start (void)
{
   int check;
   //       int address = base_port + 4;
   //       const char *testbuff=0x00;
   //       int address1 = base_port;   // * * * * * * *
   //       char *ptr;

   check = register_device (device_name, &FileOps);

   if (check<0)
     {
      printk ("<1>Can't register Device :( \n");
      return check;
     }


   check=allocate_ports(base_port,port_range);
   if (check<0)
     {

      printk ("<1>Can't load Module :( \n");
         }

   else
     {
      printk ("<1>Module loaded succesfully :) \n");
     }

   set_operating_mode();

   return check;
}

//My EXIT function * * * * * * * * * * * * * * * * * * * * *
static void __exit clean (void)
{

   unregister_device(device_name);

   release_region(base_port,port_range); //release ports

}


//Set Orerating Mode for ISA I/O card
//PortA will be an INPUT
//PortB will be an OUTPUT
static void set_operating_mode()
{

   int register_address1 = base_port + 3;
   int register_address2 = base_port + 7;

   outb_p(mod_byte2, register_address1);
   outb_p(mod_byte1, register_address2);
//   outb_p( 0x80, 0x203 );


   printk ("<1>SETOPMOD \n");

}



//Allocating Ports to the Registered Device * * * * * * * * * * * * *
* * *
static int allocate_ports (unsigned int base_port, unsigned int
port_range)
{

  int err;

   if ((err=check_region (base_port,port_range)) < 0)
     return err; // device busy

   request_region (base_port,port_range,device_name);

   return 0;
```

```
}



// REGISTER DEVICE * * * * * * * * * * * * * * * * * * * * * *
static int register_device (char *device_name, struct file_operations
*FileOps)
{
   int check;


   if (device_name == NULL)
     device_name = "greg";

   //register device with kernel and return >=0 for true || <=0 for
false
   //use the current Major Number for the specific device according
the File Operations
   check = register_chrdev(MajorNum, device_name, FileOps);

   if (check<0)
     {
      printk ("<1>Can't register device :( \n");
      return check;
     }
   return 0;
}

// UNREGISTER DEVICE * * * * * * * * * * * * * * * * * * * * * *
static int unregister_device (char *device_name)
{

   int check;
   check = unregister_chrdev(MajorNum, device_name);

   if (check<0)
     {
      printk ("<1>Can't unregister device.... Maybe in use...\n");
      return check;
     }
   return 0;
}


// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * *
// FILE OPERATIONS
// OPEN
// READ
// WRITE
// RELEASE
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * *
//
//
//
static int open_card (struct inode *inode, struct file *file)
{
   printk ("<1>OPEN \n");
   if(MOD_IN_USE>0)
```

```
     return -EBUSY;  //if the card is beeing accesed already return
"Device or Resource Busy"
   //increment usage count
   MOD_INC_USE_COUNT;
   printk ("<1>OPEN2 \n");
   return 0;
}


static int close_card (struct inode *inode, struct file *file)
{

   //decrement usage count
   MOD_DEC_USE_COUNT;
   printk ("<1>CLOSE");
   return 0;
}


static int write_card (struct file *filp, const char *ptr, size_t
count, loff_t *f_pos  )
{
   int address = base_port;
   int buff = *ptr;

   outb(buff, address );   //write to device

   return 0;

}


static int read_card (struct file *filp, char *ptr, size_t count,
loff_t *f_pos)
{
   unsigned char address =  base_port + 4;   // * * * * * * *
   int n = count;
   while (n--)
     {
      printk ("<1>READ");
      *(ptr++) = inb_p(address);  // get data from device
        printk ("<1>READEN %d from device",*ptr);
     }
   printk ("<1>READ");
   return (int)count;
}
```

The following script files were used to compile the module and the user – space program.

compile the user space program
sercom-alt.sh

```
#!/bin/bash

gcc -Wall user-alt.c -o user-alt.exe
```

compile the module
compile-alt.sh

```
#!/bin/bash

gcc -D __KERNEL__ -O2  -Wall -I"/usr/src/linux/include/" -c labcard-
alt.c -o labcard-alt.o
```

Linux Device Drivers

Progress Report of Level 3 Project
21st – 28th March 2003

Review Tasks Set Last Week

- Testing the source code of the module.
- Finishing the writing of the second sub-report.

Tasks Set This Week

- All functions of the module work. The write function works perfect and it has been tested also using Leds. The read function can read values from the ISA card but the testing will be performed during the 3rd stage of the project.
- Combine all the work that has been done up to date in order to finish and submit the final version of the second sub-report at the end of this week.

Forward look to tasks for next stage

- The final testing of the module and the writing of the third sub-report will start the next week.

Linux Device Drivers

Progress Report of Level 3 Project
28th – 25th April 2003

Review Tasks Set Last Week

- Testing the source code of the module.
- Combining all the work and research has been done up to date.
- Started writing the third and final report.

Tasks Set This Week

- The final report of the project is in a very good level.
- The previous work from the first and the second sub – report has been attached to the final report.
- A few more tests were performed to the module.
- Led lights are used to test the ISA I/O card

Forward look to tasks for next stage

- Finishing the final report and solve the problem that the read function has reading the wrong value.

Linux Device Drivers

Progress Report of Level 3 Project
<u>25th – 2nd May 2003</u>

Review Tasks Set Last Week

- Testing the source code of the module.
- Continue on writing the final report.
- E-mail the manufacturer of the I/O card in order to check if the card has a hardware problem.

Tasks Set This Week

- The final report of the project is constantly updated and in a very good level.
- A few more tests were performed to the module.
- There was no answer from the manufacturer of the I/O card and we are still trying to solve the problem.

Forward look to tasks for next stage

- Bring the final report to the last state of writing and try to find out if there could be any solution about the I/O card in order it could read correctly.

Linux Device Drivers

Progress Report of Level 3 Project
2nd – 9th May 2003

Review Tasks Set Last Week

- Testing the source code of the module.
- Continue on writing the final report.

Tasks Set This Week

- The final report of the project is constantly updated and in a very good level.
- An extension was given to the final submission of the final year project by the office of school of computing. The new date of submission is Wednesday 21 May 2003 at 12:30pm.
- Approximately, according the progress has been made; the project will be ready for submission on Monday 19th of May.

Forward look to tasks for next stage

- Finishing of the final report.

Linux Device Drivers

Progress Report of Level 3 Project
<u>9th – 16th May 2003</u>

Review Tasks Set Last Week

- Finishing the final report

Tasks Set This Week

- Inform the supervisor that the final repost has finished.
- Submission of the final report

Forward look to tasks for next stage

- None

Objectives Settings Report

<u>FINAL YEAR DEGREE PROJECT</u>

<u>**Objective Settings Proforma**</u>

<u>(to be completed and submitted by end of week four, Autumn Term)</u>

Student's Name:	Grigorios Fragkos........................
First Assessor:	Dr. Gauis Mulley.........................
Second Assessor:	Mr. Keith Verheyden…………....
Project Title:	Linux Device Drivers................. ....
Project Objectives:	General research on Linux.......... Research device drivers............... Research on "how the ISA card works" ....
Deliverables:	Implementation: …..................... - Kernel Device Driver for the ISA I/O card................................ .... - User mode test program............ ....

Please tick this box to indicate your awareness of the university's policy on ethical issues

ü

The deliverables and objectives can often change due to unforeseen circumstances, or through the student's research causing the project to follow a different path. If this is the case, and the project objectives change significantly, then the first assessor should make a note of the date and fill in a new objectives proforma, which should also be included as an appendix to the project report. The project organiser is to be consulted at this stage.

Agreed Marking Scheme Weightings

Degree Scheme in Computing

FINAL Project Assessment and Comment Form

Student **Grigorios Fragkos......................................**

Project Title: **Linux Device Drivers.................................**

Supervisor: (1/2) **Dr. Gaius Mulley, Mr Keith Verheyden...**

Mark Category	Weighting Ranges	Agreed Weighting	Mark Allocated
Project Management	50 - 80	**50**	
Originality & Self-Direction	30 – 60	**60**	
Technical Complexity	20 – 80	**80**	
Solutions, Evaluation & Conclusions	40 – 80	**60**	
Final & Sub- Report Quality	50	**50**	
Prototype / System Demo Or Project Deliverable	40 – 60	**60**	
Sponsor Mark	00 – 60	**0**	
Sub-Total Marks	-------		
Sub-Total Percentage	-------	**50%**	%
Research (Milestone 1)	-------	**15%**	%
Design & Development Progress (Milestone 2)	-------	**15%**	%
Final Presentation	-------	**20%**	%
TOTAL PERCENTAGE	-------	**100%**	%

<u>Research (Milestone One)</u> …………………………………………………………………………………………………… …………………………………………………………………………………………………… …………………………………………………………………………………………………… …………………………………………………………………………………………………… …………………………………………………………………………………………………… ……………………………
<u>Design & Development (Milestone Two)</u> …………………………………………………………………………………………………… …………………………………………………………………………………………………… …………………………………………………………………………………………………… …………………………………………………………………………………………………… …………………………………………………………………………………………………… ……………………………
<u>Presentation/Viva:</u> …………………………………………………………………………………………………… …………………………………………………………………………………………………… …………………………………………………………………………………………………… …………………………………………………………………………………………………… …………………………………………………………………………………………………… …………………………………………………………………………………………………… …………………
<u>Student's Conclusion & Evaluation (Milestone Three)</u> …………………………………………………………………………………………………… …………………………………………………………………………………………………… …………………………………………………………………………………………………… …………………………………………………………………………………………………… …………………………………………………………………………………………………… ……………………………
<u>Overall Supervisor Comments:</u> …………………………………………………………………………………………………… …………………………………………………………………………………………………… …………………………………………………………………………………………………… …………………………………………………………………………………………………… …………………………………………………………………………………………………… ……………………………

All References:

Bibliography

[1]Daniel P. Bovet, Marco Cesati 2000. *Understanding the Linux Kernel. O'Reilly.*

[2]Bill Ball 1998. *Teach Yourself Linux in 24 Hours: Second Edition. Sams Publishing.*

[3]Alessandro Rubini & Jonathan Corbet 2001. *Linux Device Drivers: Second Edition. O'Reilly.*

[4]Matt Welsh, Matthias Kalle Dalheimer, Lar Kaufman 1999. *Running Linux. O'Reilly.*

[5]Richard Stones & Neil Matthew 2001. *Beginning Linux Programming.Wrox.*

[6]Jesse Liberty, David Hovarth 2000. *Teach Yourself C++ for Linux in 21 Days. Sams.*

[7]Don Anderson, Tom Shanley 1995. *ISA System Architecture. Addison Wesley*

[8] Dr. Andrew Blyth, Lecture Notes 2002, *Introduction to UNIX security*, MSc Information Systems and Network Security, University of Glamorgan.

[9]Linux Journal, October 2001, issue 90, How to write a Linux USB device driver p.24.

[10]Linux Magazine, September 2002, issue 23, language of C p.66.

[11]Advantech. User's Manual for the ISA card PCL-731. 48-bit Digital I/O card.

Webliography

[1]http://www.cse.ogi.edu/class/cse521/2002fall/Lec02-ISA.ppt

[2]http://sunsite.tut.fi/hwb/co_ISA_Tech.html

[3]http://www.cs.arizona.edu/computer.help/policy/DIGITAL_unix/AA-Q0R6C-TET1_html/TITLE.html

[4]http://database.sarang.net/study/linux/johnsonm/devices.html

[5]http://www.freeos.com/articles/2677/2/13/

[6]http://www.linuxplanet.com/linuxplanet/tutorials/1019/1/

[7]http://usb.cs.tum.edu/usbdoc/

[8]http://class.et.byu.edu/eet343/Lecture%20Notes/OS_DeviceDrivers.htm