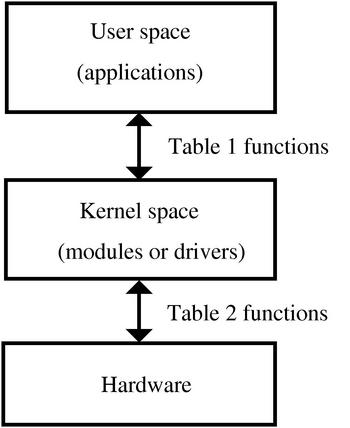
**User space and kernel space**

When you write device drivers, it’s important to make the distinction between “user space” and “kernel space”.

* **Kernel space**. Linux (which is a kernel) manages the machine's hardware in a simple and efficient manner, offering the user a simple and uniform programming interface. In the same way, the kernel, and in particular its device drivers, form a bridge or interface between the end-user/programmer and the hardware. Any subroutines or functions forming part of the kernel (modules and device drivers, for example) are considered to be part of kernel space.
* **User space**. End-user programs, like the UNIX shell or other GUI based applications (kpresenter for example), are part of the user space. Obviously, these applications need to interact with the system's hardware . However, they don’t do so directly, but through the kernel supported functions.

All of this is shown in figure 1.

[](http://www.freesoftwaremagazine.com/files/nodes/1238/spaces.jpg)Figure 1: User space where applications reside, and kernel space where modules or device drivers reside

**Interfacing functions between user space and kernel space**

The kernel offers several subroutines or functions in user space, which allow the end-user application programmer to interact with the hardware. Usually, in UNIX or Linux systems, this dialogue is performed through functions or subroutines in order to read and write files. The reason for this is that in Unix devices are seen, from the point of view of the user, as files.

On the other hand, in kernel space Linux also offers several functions or subroutines to perform the low level interactions directly with the hardware, and allow the transfer of information from kernel to user space.

Usually, for each function in user space (allowing the use of devices or files), there exists an equivalent in kernel space (allowing the transfer of information from the kernel to the user and vice-versa).

**Interfacing functions between kernel space and the hardware device**

There are also functions in kernel space which control the device or exchange information between the kernel and the hardware.

**The first driver: loading and removing the driver in user space**

I’ll now show you how to develop your first Linux device driver, which will be introduced in the kernel as a module.

For this purpose I’ll write the following program in a file named nothing.c

*<nothing.c> =*

#include <linux/module.h>

MODULE\_LICENSE("Dual BSD/GPL");

Since the release of kernel version 2.6.x, compiling modules has become slightly more complicated. First, you need to have a complete, compiled kernel source-code-tree. If you have a Debian Sarge system, you can follow the steps in Appendix B (towards the end of this article). In the following, I’ll assume that a kernel version 2.6.8 is being used.

Next, you need to generate a makefile. The makefile for this example, which should be named Makefile, will be:

*<Makefile1> =*

obj-m := nothing.o

Unlike with previous versions of the kernel, it’s now also necessary to compile the module using the same kernel that you’re going to load and use the module with. To compile it, you can type:

$ make -C /usr/src/kernel-source-2.6.8 M=pwd modules

This extremely simple module belongs to kernel space and will form part of it once it’s loaded.

In user space, you can load the module as root by typing the following into the command line:

# insmod nothing.ko

The insmod command allows the installation of the module in the kernel. However, this particular module isn’t of much use.

It is possible to check that the module has been installed correctly by looking at all installed modules:

# lsmod

Finally, the module can be removed from the kernel using the command:

# rmmod nothing

By issuing the lsmod command again, you can verify that the module is no longer in the kernel.

**The “Hello world” driver: loading and removing the driver in kernel space**

When a module device driver is loaded into the kernel, some preliminary tasks are usually performed like resetting the device, reserving RAM, reserving interrupts, and reserving input/output ports, etc.

These tasks are performed, in kernel space, by two functions which need to be present (and explicitly declared): module\_init and module\_exit; they correspond to the user space commands insmodand rmmod , which are used when installing or removing a module. To sum up, the user commandsinsmod and rmmod use the kernel space functions module\_init and module\_exit.

Let’s see a practical example with the classic program Hello world:

*<hello.c> =*

#include <linux/init.h>

#include <linux/module.h>

#include <linux/kernel.h>

MODULE\_LICENSE("Dual BSD/GPL");

static int hello\_init(void) {

printk("<1> Hello world!\n");

return 0;

}

static void hello\_exit(void) {

printk("<1> Bye, cruel world\n");

}

module\_init(hello\_init);

module\_exit(hello\_exit);

The actual functions hello\_init and hello\_exit can be given any name desired. However, in order for them to be identified as the corresponding loading and removing functions, they have to be passed as parameters to the functions module\_init and module\_exit.

The printk function has also been introduced. It is very similar to the well known printf apart from the fact that it only works inside the kernel. The <1> symbol shows the high priority of the message (low number). In this way, besides getting the message in the kernel system log files, you should also receive this message in the system console.

This module can be compiled using the same command as before, after adding its name into the Makefile.

*<Makefile2> =*

obj-m := nothing.o hello.o

In the rest of the article, I have left the Makefiles as an exercise for the reader. A complete Makefile that will compile all of the modules of this tutorial is shown in Appendix A.

When the module is loaded or removed, the messages that were written in the printk statement will be displayed in the system console. If these messages do not appear in the console, you can view them by issuing the dmesg command or by looking at the system log file with cat /var/log/syslog.

**The complete driver “memory”: initial part of the driver**

I’ll now show how to build a complete device driver: memory.c. This device will allow a character to be read from or written into it. This device, while normally not very useful, provides a very illustrative example since it is a complete driver; it's also easy to implement, since it doesn’t interface to a real hardware device (besides the computer itself).

To develop this driver, several new #include statements which appear frequently in device drivers need to be added:

*<memory initial> =*

/\* Necessary includes for device drivers \*/

#include <linux/init.h>

#include <linux/config.h>

#include <linux/module.h>

#include <linux/kernel.h> /\* printk() \*/

#include <linux/slab.h> /\* kmalloc() \*/

#include <linux/fs.h> /\* everything... \*/

#include <linux/errno.h> /\* error codes \*/

#include <linux/types.h> /\* size\_t \*/

#include <linux/proc\_fs.h>

#include <linux/fcntl.h> /\* O\_ACCMODE \*/

#include <asm/system.h> /\* cli(), \*\_flags \*/

#include <asm/uaccess.h> /\* copy\_from/to\_user \*/

MODULE\_LICENSE("Dual BSD/GPL");

/\* Declaration of memory.c functions \*/

int memory\_open(struct inode \*inode, struct file \*filp);

int memory\_release(struct inode \*inode, struct file \*filp);

ssize\_t memory\_read(struct file \*filp, char \*buf, size\_t count, loff\_t \*f\_pos);

ssize\_t memory\_write(struct file \*filp, char \*buf, size\_t count, loff\_t \*f\_pos);

void memory\_exit(void);

int memory\_init(void);

/\* Structure that declares the usual file \*/

/\* access functions \*/

struct file\_operations memory\_fops = {

read: memory\_read,

write: memory\_write,

open: memory\_open,

release: memory\_release

};

/\* Declaration of the init and exit functions \*/

module\_init(memory\_init);

module\_exit(memory\_exit);

/\* Global variables of the driver \*/

/\* Major number \*/

int memory\_major = 60;

/\* Buffer to store data \*/

char \*memory\_buffer;

After the #include files, the functions that will be defined later are declared. The common functions which are typically used to manipulate files are declared in the definition of the file\_operationsstructure. These will also be explained in detail later. Next, the initialization and exit functions—used when loading and removing the module—are declared to the kernel. Finally, the global variables of the driver are declared: one of them is the major number of the driver, the other is a pointer to a region in memory,memory\_buffer, which will be used as storage for the driver data.

**The “memory” driver: connection of the device with its files**

In UNIX and Linux, devices are accessed from user space in exactly the same way as files are accessed. These device files are normally subdirectories of the /dev directory.

To link normal files with a kernel module two numbers are used: major number and minor number. The major number is the one the kernel uses to link a file with its driver. The minor number is for internal use of the device and for simplicity it won’t be covered in this article.

To achieve this, a file (which will be used to access the device driver) must be created, by typing the following command as root:

# mknod /dev/memory c 60 0

In the above, c means that a char device is to be created, 60 is the major number and 0 is theminor number.

Within the driver, in order to link it with its corresponding /dev file in kernel space, theregister\_chrdev function is used. It is called with three arguments: major number, a string of characters showing the module name, and a file\_operations structure which links the call with the file functions it defines. It is invoked, when installing the module, in this way:

*<memory init module> =*

int memory\_init(void) {

int result;

/\* Registering device \*/

result = register\_chrdev(memory\_major, "memory", &memory\_fops);

if (result < 0) {

printk(

"<1>memory: cannot obtain major number %d\n", memory\_major);

return result;

}

/\* Allocating memory for the buffer \*/

memory\_buffer = kmalloc(1, GFP\_KERNEL);

if (!memory\_buffer) {

result = -ENOMEM;

goto fail;

}

memset(memory\_buffer, 0, 1);

printk("<1>Inserting memory module\n");

return 0;

fail:

memory\_exit();

return result;

}

Also, note the use of the kmalloc function. This function is used for memory allocation of the buffer in the device driver which resides in kernel space. Its use is very similar to the well known malloc function. Finally, if registering the major number or allocating the memory fails, the module acts accordingly.

**The “memory” driver: removing the driver**

In order to remove the module inside the memory\_exit function, the function unregsiter\_chrdevneeds to be present. This will free the major number for the kernel.

*<memory exit module> =*

void memory\_exit(void) {

/\* Freeing the major number \*/

unregister\_chrdev(memory\_major, "memory");

/\* Freeing buffer memory \*/

if (memory\_buffer) {

kfree(memory\_buffer);

}

printk("<1>Removing memory module\n");

}

The buffer memory is also freed in this function, in order to leave a clean kernel when removing the device driver.

**The “memory” driver: opening the device as a file**

The kernel space function, which corresponds to opening a file in user space (fopen), is the memberopen: of the file\_operations structure in the call to register\_chrdev. In this case, it is thememory\_open function. It takes as arguments: an inode structure, which sends information to the kernel regarding the major number and minor number; and a file structure with information relative to the different operations that can be performed on a file. Neither of these functions will be covered in depth within this article.

When a file is opened, it’s normally necessary to initialize driver variables or reset the device. In this simple example, though, these operations are not performed.

The memory\_open function can be seen below:

*<memory open> =*

int memory\_open(struct inode \*inode, struct file \*filp) {

/\* Success \*/

return 0;

}

**The “memory” driver: closing the device as a file**

The corresponding function for closing a file in user space (fclose) is the release: member of thefile\_operations structure in the call to register\_chrdev. In this particular case, it is the functionmemory\_release, which has as arguments an inode structure and a file structure, just like before.

When a file is closed, it’s usually necessary to free the used memory and any variables related to the opening of the device. But, once again, due to the simplicity of this example, none of these operations are performed.

The memory\_release function is shown below:

*<memory release> =*

int memory\_release(struct inode \*inode, struct file \*filp) {

/\* Success \*/

return 0;

}

This new function is shown in Table 6.

**Table 6. Device driver events and their associated interfacing functions between kernel space and user space.**

**The “memory” driver: reading the device**

To read a device with the user function fread or similar, the member read: of thefile\_operations structure is used in the call to register\_chrdev. This time, it is the functionmemory\_read. Its arguments are: a type file structure; a buffer (buf), from which the user space function (fread) will read; a counter with the number of bytes to transfer (count), which has the same value as the usual counter in the user space function (fread); and finally, the position of where to start reading the file (f\_pos).

In this simple case, the memory\_read function transfers a single byte from the driver buffer (memory\_buffer) to user space with the function copy\_to\_user:

*<memory read> =*

ssize\_t memory\_read(struct file \*filp, char \*buf,

size\_t count, loff\_t \*f\_pos) {

/\* Transfering data to user space \*/

copy\_to\_user(buf,memory\_buffer,1);

/\* Changing reading position as best suits \*/

if (\*f\_pos == 0) {

\*f\_pos+=1;

return 1;

} else {

return 0;

}

}

The reading position in the file (f\_pos) is also changed. If the position is at the beginning of the file, it is increased by one and the number of bytes that have been properly read is given as a return value, 1. If not at the beginning of the file, an end of file (0) is returned since the file only stores one byte.

**The “memory” driver: writing to a device**

To write to a device with the user function fwrite or similar, the member write: of thefile\_operations structure is used in the call to register\_chrdev. It is the functionmemory\_write, in this particular example, which has the following as arguments: a type file structure;buf, a buffer in which the user space function (fwrite) will write; count, a counter with the number of bytes to transfer, which has the same values as the usual counter in the user space function (fwrite); and finally, f\_pos, the position of where to start writing in the file.

*<memory write> =*

ssize\_t memory\_write( struct file \*filp, char \*buf,

size\_t count, loff\_t \*f\_pos) {

char \*tmp;

tmp=buf+count-1;

copy\_from\_user(memory\_buffer,tmp,1);

return 1;

}

In this case, the function copy\_from\_user transfers the data from user space to kernel space.

In Table 8 this new function is shown.

|  |  |  |
| --- | --- | --- |
| Events | User functions | Kernel functions |
| Load module | insmod | module\_init() |
| Open device | fopen | file\_operations: open |
| Close device | fread | file\_operations: read |
| Write device | fwrite | file\_operations: write |
| Close device | fclose | file\_operations: release |
| Remove module | rmmod | module\_exit() |

**Device driver events and their associated interfacing functions between kernel space and user space.**

**The complete “memory” driver**

By joining all of the previously shown code, the complete driver is achieved:

*<memory.c> =*

<memory initial>

<memory init module>

<memory exit module>

<memory open>

<memory release>

<memory read>

<memory write>

Before this module can be used, you will need to compile it in the same way as with previous modules. The module can then be loaded with:

# insmod memory.ko

It’s also convenient to unprotect the device:

# chmod 666 /dev/memory

If everything went well, you will have a device /dev/memory to which you can write a string of characters and it will store the last one of them. You can perform the operation like this:

$ echo -n abcdef >/dev/memory

To check the content of the device you can use a simple cat:

$ cat /dev/memory

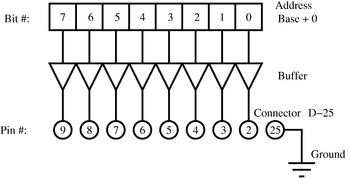
The stored character will not change until it is overwritten or the module is removed.

**The real “parlelport” driver: description of the parallel port**

I’ll now proceed by modifying the driver that I just created to develop one that does a real task on a real device. I’ll use the simple and ubiquitous computer parallel port and the driver will be called parlelport.

The parallel port is effectively a device that allows the input and output of digital information. More specifically it has a female D-25 connector with twenty-five pins. Internally, from the point of view of the CPU, it uses three bytes of memory. In a PC, the base address (the one from the first byte of the device) is usually 0x378. In this basic example, I’ll use just the first byte, which consists entirely of digital outputs.

The connection of the above-mentioned byte with the external connector pins is shown in figure 2.

[](http://www.freesoftwaremagazine.com/files/nodes/1238/pin_parport.jpg)Figure 2: The first byte of the parallel port and its pin connections with the external female D-25 connector

**The “parlelport” driver: initializing the module**

The previous memory\_init function needs modification—changing the RAM memory allocation for the reservation of the memory address of the parallel port (0x378). To achieve this, use the function for checking the availability of a memory region (check\_region), and the function to reserve the memory region for this device (request\_region). Both have as arguments the base address of the memory region and its length. The request\_region function also accepts a string which defines the module.

*<parlelport modified init module> =*

/\* Registering port \*/

port = check\_region(0x378, 1);

if (port) {

printk("<1>parlelport: cannot reserve 0x378\n");

result = port;

goto fail;

}

request\_region(0x378, 1, "parlelport");

**The “parlelport” driver: removing the module**

It will be very similar to the memory module but substituting the freeing of memory with the removal of the reserved memory of the parallel port. This is done by the release\_region function, which has the same arguments as check\_region.

*<parlelport modified exit module> =*

/\* Make port free! \*/

if (!port) {

release\_region(0x378,1);

}

**The “parlelport” driver: reading the device**

In this case, a real device reading action needs to be added to allow the transfer of this information to user space. The inb function achieves this; its arguments are the address of the parallel port and it returns the content of the port.

*<parlelport inport> =*

/\* Reading port \*/

parlelport\_buffer = inb(0x378);

Table 9 (the equivalent of Table 2) shows this new function.

|  |  |
| --- | --- |
| Events | Kernel functions |
| Read data | inb |
| Write data |  |

**Device driver events and their associated functions between kernel space and the hardware device.**

**The “parlelport” driver: writing to the device**

Again, you have to add the “writing to the device” function to be able to transfer later this data to user space. The function outb accomplishes this; it takes as arguments the content to write in the port and its address.

*<parlelport outport> =*

/\* Writing to the port \*/

outb(parlelport\_buffer,0x378);

Table 10 summarizes this new function.

|  |  |
| --- | --- |
| Events | Kernel functions |
| Read data | inb |
| Write data | outb |

**Device driver events and their associated functions between kernel space and the hardware device.**

**The complete “parlelport” driver**

I’ll proceed by looking at the whole code of the parlelport module. You have to replace the wordmemory for the word parlelport throughout the code for the memory module. The final result is shown below:

*<parlelport.c> =*

<parlelport initial>

<parlelport init module>

<parlelport exit module>

<parlelport open>

<parlelport release>

<parlelport read>

<parlelport write>

**Initial section**

In the initial section of the driver a different major number is used (61). Also, the global variablememory\_buffer is changed to port and two more #include lines are added: ioport.h andio.h.

*<parlelport initial> =*

/\* Necessary includes for drivers \*/

#include <linux/init.h>

#include <linux/config.h>

#include <linux/module.h>

#include <linux/kernel.h> /\* printk() \*/

#include <linux/slab.h> /\* kmalloc() \*/

#include <linux/fs.h> /\* everything... \*/

#include <linux/errno.h> /\* error codes \*/

#include <linux/types.h> /\* size\_t \*/

#include <linux/proc\_fs.h>

#include <linux/fcntl.h> /\* O\_ACCMODE \*/

#include <linux/ioport.h>

#include <asm/system.h> /\* cli(), \*\_flags \*/

#include <asm/uaccess.h> /\* copy\_from/to\_user \*/

#include <asm/io.h> /\* inb, outb \*/

MODULE\_LICENSE("Dual BSD/GPL");

/\* Function declaration of parlelport.c \*/

int parlelport\_open(struct inode \*inode, struct file \*filp);

int parlelport\_release(struct inode \*inode, struct file \*filp);

ssize\_t parlelport\_read(struct file \*filp, char \*buf,

size\_t count, loff\_t \*f\_pos);

ssize\_t parlelport\_write(struct file \*filp, char \*buf,

size\_t count, loff\_t \*f\_pos);

void parlelport\_exit(void);

int parlelport\_init(void);

/\* Structure that declares the common \*/

/\* file access fcuntions \*/

struct file\_operations parlelport\_fops = {

read: parlelport\_read,

write: parlelport\_write,

open: parlelport\_open,

release: parlelport\_release

};

/\* Driver global variables \*/

/\* Major number \*/

int parlelport\_major = 61;

/\* Control variable for memory \*/

/\* reservation of the parallel port\*/

int port;

module\_init(parlelport\_init);

module\_exit(parlelport\_exit);

**Module init**

In this module-initializing-routine I’ll introduce the memory reserve of the parallel port as was described before.

*<parlelport init module> =*

int parlelport\_init(void) {

int result;

/\* Registering device \*/

result = register\_chrdev(parlelport\_major, "parlelport",

&parlelport\_fops);

if (result < 0) {

printk(

"<1>parlelport: cannot obtain major number %d\n",

parlelport\_major);

return result;

}

<parlelport modified init module>

printk("<1>Inserting parlelport module\n");

return 0;

fail:

parlelport\_exit();

return result;

}

**Removing the module**

This routine will include the modifications previously mentioned.

*<parlelport exit module> =*

void parlelport\_exit(void) {

/\* Make major number free! \*/

unregister\_chrdev(parlelport\_major, "parlelport");

<parlelport modified exit module>

printk("<1>Removing parlelport module\n");

}

**Opening the device as a file**

This routine is identical to the memory driver.

*<parlelport open> =*

int parlelport\_open(struct inode \*inode, struct file \*filp) {

/\* Success \*/

return 0;

}

**Closing the device as a file**

Again, the match is perfect.

*<parlelport release> =*

int parlelport\_release(struct inode \*inode, struct file \*filp) {

/\* Success \*/

return 0;

}

**Reading the device**

The reading function is similar to the memory one with the corresponding modifications to read from the port of a device.

*<parlelport read> =*

ssize\_t parlelport\_read(struct file \*filp, char \*buf,

size\_t count, loff\_t \*f\_pos) {

/\* Buffer to read the device \*/

char parlelport\_buffer;

<parlelport inport>

/\* We transfer data to user space \*/

copy\_to\_user(buf,&parlelport\_buffer,1);

/\* We change the reading position as best suits \*/

if (\*f\_pos == 0) {

\*f\_pos+=1;

return 1;

} else {

return 0;

}

}

**Writing to the device**

It is analogous to the memory one except for writing to a device.

*<parlelport write> =*

ssize\_t parlelport\_write( struct file \*filp, char \*buf,

size\_t count, loff\_t \*f\_pos) {

char \*tmp;

/\* Buffer writing to the device \*/

char parlelport\_buffer;

tmp=buf+count-1;

copy\_from\_user(&parlelport\_buffer,tmp,1);

<parlelport outport>

return 1;

}

**LEDs to test the use of the parallel port**

In this section I’ll detail the construction of a piece of hardware that can be used to visualize the state of the parallel port with some simple LEDs.

**WARNING: Connecting devices to the parallel port can harm your computer. Make sure that you are properly earthed and your computer is turned off when connecting the device. Any problems that arise due to undertaking these experiments is your sole responsibility.**

The circuit to build is shown in figure 3 You can also read “PC & Electronics: Connecting Your PC to the Outside World” by Zoller as reference.

In order to use it, you must first ensure that all hardware is correctly connected. Next, switch off the PC and connect the device to the parallel port. The PC can then be turned on and all device drivers related to the parallel port should be removed (for example, lp, parport, parport\_pc, etc.). The hotplugmodule of the Debian Sarge distribution is particularly annoying and should be removed. If the file/dev/parlelport does not exist, it must be created as root with the command:

# mknod /dev/parlelport c 61 0

Then it needs to be made readable and writable by anybody with:

# chmod 666 /dev/parlelport

The module can now be installed, parlelport. You can check that it is effectively reserving the input/output port addresses 0x378 with the command:

$ cat /proc/ioports

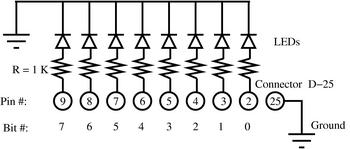
To turn on the LEDs and check that the system is working, execute the command:

$ echo -n A >/dev/parlelport

This should turn on LED zero and six, leaving all of the others off.

You can check the state of the parallel port issuing the command:

$ cat /dev/parlelport

[](http://www.freesoftwaremagazine.com/files/nodes/1238/leds.jpg)Figure 3: Electronic diagram of the LED matrix to monitor the parallel port

**Final application: flashing lights**

Finally, I’ll develop a pretty application which will make the LEDs flash in succession. To achieve this, a program in user space needs to be written with which only one bit at a time will be written to the/dev/parlelport device.

*<lights.c> =*

#include <stdio.h>

#include <unistd.h></p>

int main() {

unsigned char byte,dummy;

FILE \* PARLELPORT;

/\* Opening the device parlelport \*/

PARLELPORT=fopen("/dev/parlelport","w");

/\* We remove the buffer from the file i/o \*/

setvbuf(PARLELPORT,&dummy,\_IONBF,1);

/\* Initializing the variable to one \*/

byte=1;

/\* We make an infinite loop \*/

while (1) {

/\* Writing to the parallel port \*/

/\* to turn on a LED \*/

printf("Byte value is %d\n",byte);

fwrite(&byte,1,1,PARLELPORT);

sleep(1);

/\* Updating the byte value \*/

byte<<=1;

if (byte == 0) byte = 1;

}

fclose(PARLELPORT);

}

It can be compiled in the usual way:

$ gcc -o lights lights.c

and can be executed with the command:

$ lights

The lights will flash successively one after the other! The flashing LEDs and the Linux computer running this program are shown in figure 4.

**Conclusion**

Having followed this brief tutorial you should now be capable of writing your own complete device driver for simple hardware like a relay board (see Appendix C), or a minimal device driver for complex hardware. Learning to understand some of these simple concepts behind the Linux kernel allows you, in a quick and easy way, to get up to speed with respect to writing device drivers. And, this will bring you another step closer to becoming a true Linux kernel developer.

**Appendix A. Complete Makefile**

*<Makefile> =*

obj-m := nothing.o hello.o memory.o parlelport.o

**Appendix B. Compiling the kernel on a Debian Sarge system**

To compile a 2.6.x kernel on a Debian Sarge system you need to perform the following steps, which should be run as root:

* Install the “kernel-image-2.6.x” package.
* Reboot the machine to make this the running kernel image. This is done semi-automatically by Debian. You may need to tweak the lilo configuration file /etc/lilo.conf and then run lilo to achieve this.
* Install the “kernel-source-2.6.x” package.
* Change to the source code directory, cd /usr/src and unzip and untar the source code withbunzip2 kernel-source-2.6.x.tar.bz2 and tar xvf kernel-source-2.6.x.tar. Change to the kernel source directory with cd /usr/src/kernel-source-2.6.x
* Copy the default Debian kernel configuration file to your local kernel source directory cp /boot/config-2.6.x .config.
* Make the kernel and the modules with make and then make modules.

**Appendix C. Exercises**

If you would like to take on some bigger challenges, here are a couple of exercises you can do:

* I once wrote two device drivers for two ISA [Meilhaus](http://meilhaus.de/) boards, an analog to digital converter (ME26) and a relay control board (ME53). The software is available from the [ADQ](http://adq.sourceforge.net/) project. Get the newer PCI versions of these Meilhaus boards and update the software.
* Take any device that doesn’t work on Linux, but has a very similar chipset to another device which does have a proven device driver for Linux. Try to modify the working device driver to make it work for the new device. If you achieve this, submit your code to the kernel and become a kernel developer yourself!

**Comments and acknowledgements**

Three years have elapsed since the [first version](http://es.tldp.org/Presentaciones/200103hispalinux/calbet/html/t1.html) of this document was written. It was originally written in Spanish and intended for version 2.2 of the kernel, but kernel 2.4 was already making its first steps at that time. The reason for this choice is that good documentation for writing device drivers, the **Linux device drivers** book (see bibliography), lagged the release of the kernel in some months. This new version is also coming out soon after the release of the new 2.6 kernel, but up to date documentation is now readily available in [Linux Weekly News](http://lwn.net/Articles/driver-porting) making it possible to have this document synchronized with the newest kernel.

Fortunately enough, PCs still come with a built-in parallel port, despite the actual trend of changing everything inside a PC to render it obsolete in no time. Let us hope that PCs still continue to have built-in parallel ports for some time in the future, or that at least, parallel port PCI cards are still being sold.

This tutorial has been originally typed using a text editor (i.e. emacs) in noweb format. This text is then processed with the noweb tool to create a LaTeX file ( .tex ) and the source code files ( .c ). All this can be done using the supplied makefile.document with the command make -f makefile.document.