Writing device drivers in Linux: A brief tutorial

"Do you pine for the nice days of Minix-1.1, when men were men and wrote their own device drivers?" *Linus Torvalds*

Pre-requisites

In order to develop Linux device drivers, it is necessary to have an understanding of the following:

- **C programming**. Some in-depth knowledge of C programming is needed, like pointer usage, bit manipulating functions, etc.
- **Microprocessor programming**. It is necessary to know how microcomputers work internally: memory addressing, interrupts, etc. All of these concepts should be familiar to an assembler programmer.

There are several different devices in Linux. For simplicity, this brief tutorial will only cover type `char` devices loaded as modules. Kernel 2.6.x will be used (in particular, kernel 2.6.8 under Debian Sarge, which is now Debian Stable).

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.

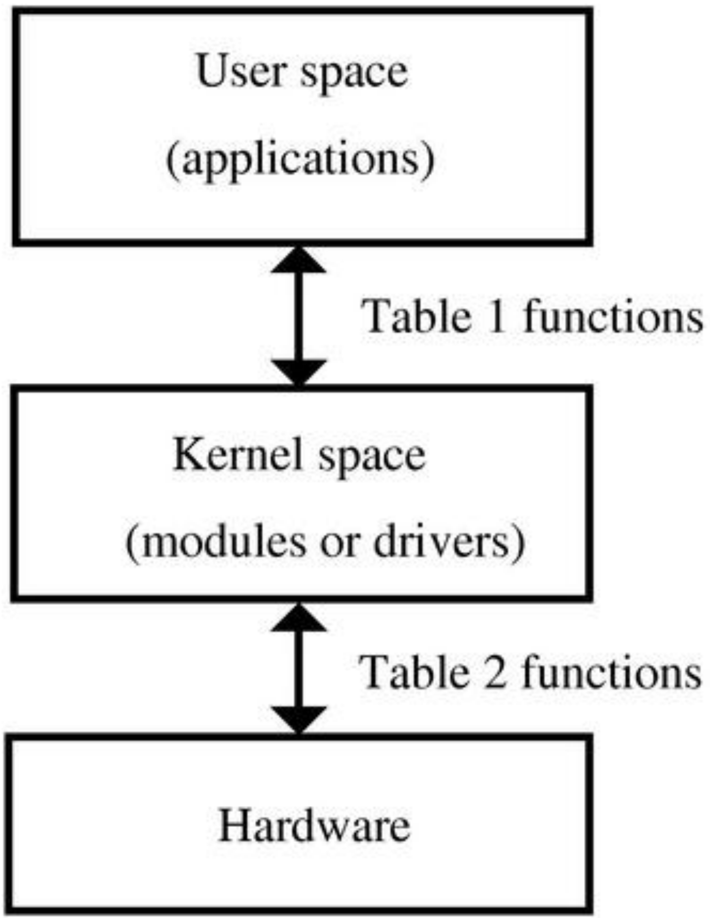

Figure 1: User space where applications reside, and kernel space where modules or device drivers reside

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). This is shown in Table 1, which is, at this point, empty. It will be filled when the different device drivers concepts are introduced.

Events	User functions	Kernel functions
Load module		
Open device		
Read device		
Write device		
Close device		
Remove module		

Table 1. Device driver events and their associated interfacing functions in

kernel space and user space.

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. Table 2 illustrates these concepts. This table will also be filled as the concepts are introduced.

Don't miss out on the other pages!
123456789…next ›last »

Events	Kernel functions
Read data	
Write data	

Table 2. Device driver events and their associated functions between kernel space and the hardware device.

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 summary of all this is shown in Table 3.

Events	User functions	Kernel functions
Load module	insmod	
Open device		
Read device		
Write device		
Close device		
Remove module	rmmod	

Table 3. Device driver events and their associated interfacing functions between kernel space and user 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 `insmod` and `rmmod` , which are used when installing or removing a module. To sum up, the user commands `insmod` 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`.

Table 4 shows these two new functions.

Events	User functions	Kernel functions
Load module	insmod	module_init()
Open device		
Read device		
Write device		
Close device		
Remove module	rmmod	module_exit()

Table 4. Device driver events and their associated interfacing functions between kernel space and user space.

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_operations` structure. 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 the `minor number`.

Within the driver, in order to link it with its corresponding `/dev` file in kernel space, the `register_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_chrdev` needs 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 member `open:` of the `file_operations` structure in the call to `register_chrdev`. In this case, it is the `memory_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;
}
```

This new function is now shown in Table 5.

Events	User functions	Kernel functions
Load module	insmod	module_init()
Open device	fopen	file_operations: open
Read device		
Write device		
Close device		
Remove module	rmmod	module_exit()

Table 5. Device driver events and their associated interfacing functions between kernel space and user space.

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 the `file_operations` structure in the call to `register_chrdev`. In this particular case, it is the function `memory_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.

Events	User functions	Kernel functions
Load module	insmod	module_init()
Open device	fopen	file_operations: open
Read device		
Write device		
Close device	fclose	file_operations: release
Remove module	rmmod	module_exit()

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 the `file_operations` structure is used in the call to `register_chrdev`. This time, it is the function `memory_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.

In Table 7 this new function has been added.

Events	User functions	Kernel functions
Load module	insmod	module_init()
Open device	fopen	file_operations: open
Read device	fread	file_operations: read
Write device		
Close device	fclose	file_operations: release
Remove modules	rmmod	module_exit()

Table 7. Device driver events and their associated interfacing functions

between kernel space and user space.

To write to a device with the user function `fwrite` or similar, the member `write:` of the `file_operations` structure is used in the call to `register_chrdev`. It is the function `memory_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.

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.

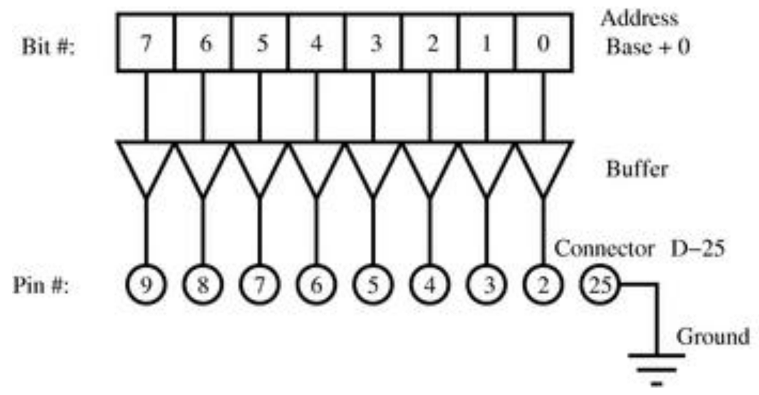

Figure 2: The first byte of the parallel port and its pin connections with the external female D-25 connector

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

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);
```