MASTER OF SCIENCE
IN ENGINEERING

Embedded Realtime Software

Linux drivers - Exercise

Scope	Learn how to implement a device driver for the Linux OS.
Keywords	Linux, driver
Prerequisites	Linux basic knowledges
Contact	Roberto Bucher, roberto.bucher@supsi.ch
Version	2009-05-19
Date	September 20, 2009

Contents

1 Basics . . . 4
2 Why a driver? . . . 5
  2.1 Devices drivers . . . 5
  2.2 “Char” devices drivers . . . 5
3 The driver for the parallel port . . . 5
4 Open the driver . . . 6
5 Close the driver . . . 6
6 Write to the device . . . 6
7 Read from the device . . . 6
8 Associate the module functions . . . 7
9 Module initialization . . . 7
10 Closing the driver . . . 7
11 Compile and install the driver . . . 8
12 Calling the driver functions from a user space program . . . 8
13 The complete driver . . . 8

1 Basics

The Linux kernel takes advantages of the possibility to write kernel drivers as modules which can be uploaded on request. A device driver allows normal users to access the part of the system that are normally protected (memory, ports etc.). This method has different advantages:

- The kernel can be highly modularized, in order to be compatible with the most possible hardware
- A kernel module can be modified without need of recompiling the full kernel

In order to write, modify, compile and upload a device driver, the user needs temporarily superuser (root) permissions.

A device driver contains at least two functions:

- A function for the module initialization (executed when the module is loaded with the command "insmod")
- A function to exit the module (executed when the module is removed with the command "rmmod")

The following code (module1.c) shows a very simple kernel module:

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

MODULE_LICENSE("GPL");

static int __init init_mod(void)
{
  printk("Module1 started...\n");
  return 0;
}

static void __exit end_mod(void)
{
  printk("Module1 ended...\n");
}

module_init(init_mod);
module_exit(end_mod);
```

The following "Makefile" can be used to compile and generate the linux module:

```
obj-m:= module1.o

KDIR = /lib/modules/$(shell uname -r)/build
PWD  = $(shell pwd)

default:
$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules

clean:
rm -f *.mod.* *.o *.ko
```

Now the module can be uploaded using the command "insmod" and removed using the command "rmmod".

```
insmod ./modulo1.ko
dmesg
rmmod modulo1
dmesg
```

The command "dmesg" allows to see the output of the modules, printed using the specific function "printk", corresponding to the function "printf" in user space. At the end of the output of "dmesg" the following lines are shown:

```
Modulo1 started
Modulo1 unloaded
```

2 Why a driver?

2.1 Devices drivers

The role of a driver is to provide mechanisms which allows normal user to access protected parts of its system, in particular ports, registers and memory addresses normally managed by the operating system.

One of the good features of Linux is the ability to extend at runtime the set of the features offered by the kernel. Users can add or remove functionalities to the kernel while the system is running.

These "programs" that can be added to the kernel at runtime are called "module". The Linux kernel offers support for different classes of modules:

- "char" devices are devices that can be accessed as a stream of bytes (like a file).
- "block" devices are accessed by filesystem nodes (example: disks).
- "network" interfaces are able to exchange data with other hosts.

2.2 "Char" devices drivers

A physical device, for example a I/O port, can be accessed like a file, using the commands "open", "read", "write" and "close" provided by the prototypes described under "fcntl.h".

First, a device must be configured in the folder "/dev", providing a class (for example "c" for a char device), a major and a minor number in order to identify it. The major number is used by the driver initialization for associating a module driver to a specific device. The most important part of the driver is represented by the "file operation" table, which associates the different operations on the device (open, read, write, close etc) to the implemented ones in the driver.

In the next section we'll write a very simple driver which writes and read a byte to and from the parallel port of a PC.

3 The driver for the parallel port

Note: The procedures described in the following have mandatory input and output parameters! Our device will be registered as "/dev/ppdrv".

4 Open the driver

This function is automatically called when the user call for example

```
fd = open(‘‘/dev/ppdrv’’, O_RDWR);
```

in its user space program.

In our example this procedure doesn’t do something particular. Only a simple message is sent to the log file.

```
static int pp_open(struct inode *inode, struct file *filep)
{
  printk("Device opened\n");
  return 0;
}
```

5 Close the driver

This procedure will be associated to the command

```
close(fd);
```

A simple message is sent to the log file.

```
static int pp_release(struct inode *inode, struct file *filep)
{
  printk("Device closed\n");
  return 0;
}
```

6 Write to the device

This function is associated to the function “write” in the user space program. Using a driver allows to access the hardware or the resource without needing to set particular permissions (for example using ioperm or iopl commands). The user can directly work with commands like “outb, inb etc.).

```
static ssize_t pp_write(struct file *filep, const char *buf, size_t count, loff_t *f_pos)
{
  printk("Writing to the device...\n");
  return 1;
}
```

7 Read from the device

This function is assigned to the user space operation ”read”.

```
static ssize_t pp_read(struct file *filep, char *buf, size_t count, loff_t *f_pos)
{
  printk("Reading from the device...\n");
  return 1;
}
```

8 Associate the module functions

The module must register the programmed functions to the device functions using the “file_operations” structure in the initializations procedure.This structure is defined in the “linux/fs.h” prototype.

```
struct file_operations pp_fops =
{
  owner:      THIS_MODULE,
  open:       pp_open,
  release:    pp_release,
  read:       pp_read,
  write:      pp_write,
};
```

9 Module initialization

The main task of the initialization procedure is to register the driver and to associate it to a specific device.

In particular, the function “register_chardev” is used to associate the written module to the device with a specific major number, the name of the device. In our example the device will be registered with the major number 22.

```
static int __init init_pp(void)
{
  int result;
  if ((result = register_chrdev(22, "ppio", &pp_fops)) < 0)
    goto fail;
  printk("PP driver loaded...\n");
  return 0;
 fail:
  return -1;
}
```

10 Closing the driver

This procedure is responsible of unregistering the driver module.

```
static void __exit end_pp(void)
{
  unregister_chrdev(22, "ppio");
  printk("PP driver unloaded...\n");
}
```

The last job is to add the following line to the module.

```
MODULE_LICENSE("GPL");
```

and these two lines at the end of the module

```
module_init(init_mod);
module_exit(end_mod);
```

11 Compile and install the driver

The following "Makefile" is used to compile the driver

```
obj-m:= ppio_drv.o

KDIR = /lib/modules/$(shell uname -r)/build
PWD  = $(shell pwd)

default:
$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules

clean:
rm -f *.mod.* *.o *.ko
```

The compiled driver can be put in the kernel area by the command

```
insmod ./ppio_drv.ko
```

A device must be created under the folder "/dev" in order to use the generated module. This device must be created as character device with the major number 22 and the minor number 0 using the following command, from a shell. This command must be run as superuser.

```
mknod -m 0666 /dev/ppio c 22 0
```

The device "/dev/ppio" is created with the as character device ("c"), with the major number 22 and the minor number 0.

12 Calling the driver functions from a user space program

The following program take advantage of the generated driver. It writes and reads a byte to and from the parallel port using the EPP protocol.

```
#include <fcntl.h>
#include <stdio.h>
int main()
{
char buffer[1];
int fd;

  fd=open("/dev/ppio",O_RDWR);
  buffer[0]=0x00;
  write(fd,buffer,1,NULL);
  read(fd,buffer,1,NULL);
  printf("Value : 0x%02x\n",buffer[0]);
  close(fd);
}
```

13 The complete driver

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

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

MODULE_LICENSE("GPL");

static int pp_open(struct inode *inode, struct file *filep)
{
  printk("Device open\n");
  return 0;
}

static int pp_release(struct inode *inode, struct file *filep)
{
  printk("Device closed\n");
  return 0;
}

static ssize_t pp_read(struct file *filep, char *buf, size_t count, loff_t *f_pos)
{
  printk("Reading...\n");
  return 0;
}

static ssize_t pp_write(struct file *filep, const char *buf, size_t count, loff_t *f_pos)
{
  printk("Writing...\n");
  return 0;
}

struct file_operations pp_fops =
{
  owner:      THIS_MODULE,
  open:       pp_open,
  release:    pp_release,
  read:       pp_read,
  write:      pp_write,
};

static int __init init_mod(void)
{
  int result;
  if ((result = register_chrdev(22, "ppio", &pp_fops)) < 0)
    goto fail;
  printk("Modulo1 started...\n");
  return 0;
 fail:
  return -1;
}

static void __exit end_mod(void)
{
  unregister_chrdev(22, "ppio");
  printk("Modulo1 ended...\n");
}
```

```
module_init(init_mod);
module_exit(end_mod);
```

Bibliography

[1] *Linux Device Drivers.* Jonathan Corbet and Alessandro Rubini and Greg Kroah-Hartman, 2006.