LINUX JOURNAL
(/)

Writing a Linux Driver

Embedded (/tag/embedded)

by Fernando Matia on March 31, 1998

The concept of an operating system (OS) must be well understood before any attempt to navigate inside it is made. Several definitions are available for an OS:

1. An OS is the set of manual and automatic procedures which allow a set of users to share a computing system in an efficient manner.
2. The dictionary defines an OS as a program or set of programs which manage the processes of a computing system and allow the normal execution of the other jobs.
3. The definition from the Tanenbaum book (see Resources): An operating system is [the program] which controls all the resources of the computer and offers the support where users can develop application programs.

It is also very important to clearly distinguish a *program* from a *process*. A program is a block of data plus instructions, which is stored in a file on disk and is ready to be executed. On the other hand, a process is an image in memory of the program which is being executed. This difference is highly important, because usually the processes are running under OS control. Here, our program is the OS, so we cannot speak about processes.

We will use the term *kernel* to refer to the main body of the OS, which is a program written in the C language. The program file may be named vmlinuz, vmlinux or zImage, and has some things in common with the MS-DOS files COMMAND.COM, MSDOS.SYS and IO.SYS, although their functionality is different. When we discuss *compilation* of the kernel, we mean that we will edit the source files in order to generate a new kernel.

Peripheral or internal *devices* allow users to communicate with the computer. Examples of devices are: keyboards, monitors, floppy and hard disks, CD-ROMs, printers, mice (serial/parallel), networks, modems, etc. A *driver* is the part of the OS that manages communication with devices; thus, they are usually called *device drivers*.
What is a Driver?

Figure 1. Software/Hardware Scheme
Figure 1 shows the relation between user programs, the OS and the devices. Differences between software and hardware are clearly specified in this scheme. At the left side, user programs may interact with the devices (for example, a hard disk) through a set of high-level library functions. For example, we can open and write to a file of the hard disk calling the C library functions **fopen**, **fprintf** and **close**:

```
FILE *fid=fopen("filename", "w");
fprintf(fid, "Hello, world!");
fclose(fid);
```

The user can also write to a file (or to another device such as a printer) from the OS shell, using commands such as:

```
echo "Hello, world!" >
echo "Hello, world!" > /dev/lp
```

To execute this command, both the shell and the library functions perform a call to a low level function of the OS, e.g., **open()**, **write()** or **close()**:

```
fid = open("/dev/lp", O_WRONLY);
write(fid, "Hello, world!");
close(fid);
```

Each device can be referred to as a special file named /dev/*. Internally, the OS is composed of a set of drivers, which are pieces of software that perform the low-level communication with each device. At this execute level, the kernel calls driver functions such as **lp_open()** or **lp_write()**.

On the right side of Figure 1, the hardware is composed of the device (a video display or an Ethernet link) plus an interface (a VGA card or a network card). Finally, the device driver is the physical interface between the software and the hardware. The driver reads from and writes to the hardware through ports (memory addresses where the hardware links physically), using the internal functions **out_p** and **in_p**:

```
out_p(0x3a, 0x1f);
data = in_p(0x3b);
```

Note that these functions are not available to the user. Since the Linux kernel runs in protected mode, the low memory addresses, where the ports addresses reside, are not user accessible. Functions equivalent to the low-level functions **in** and **out** do not exist in the high-level library, as in other operating systems such as MS-DOS.

Features of a Driver

The main features of a driver are:

- It performs input/output (I/O) management.
- It provides transparent device management, avoiding low-level programming (ports).
- It increases I/O speed, because usually it has been optimized.
- It includes software and hardware error management.
- It allows concurrent access to the hardware by several processes.

There are four types of drivers: character drivers, block drivers, terminal drivers and streams. *Character drivers* transmit information from the user to the device (or vice versa) byte per byte (see Figure 2). Two examples are the printer, /dev/lp, and the memory (yes, the memory is also a device), /dev/mem.

Figure 2. Character Drivers

Block drivers (see Figure 3) transmit information block per block. This means that the incoming data (from the user or from the device) are stored in a buffer until the buffer is full. When this occurs, the buffer content is physically sent to the device or to the user. This is the reason why all the printed messages do not appear in the screen when a user program crashes (the messages in the buffer were lost), or the floppy drive light does not always turn on when the user writes to a file. The clearest examples of this type of driver are disks: floppy disks (/dev/fd0), IDE hard disks (/dev/hda) and SCSI hard disks (/dev/sd1).

Figure 3. Block Drivers

Terminal drivers (see Figure 4) constitute a special set of character drivers for user communication. For example, command tools in an open windows environment, an X terminal or a console, are devices which require special functions, e.g., the up and down arrows for a command buffer manager or tabbing in the bash shell. Examples of block drivers are /dev/tty0 or /dev/ttya (a serial port). In both cases the kernel includes special routines, and the driver special procedures, to cope with all particular features.

Figure 4. Terminal Drivers

Streams are the youngest drivers (see Figure 5) and are designed for very high speed data flows. Both the kernel and the driver include several protocol layers. The best example of this type is a network driver.

Figure 5. Stream Drivers

As we have said, a driver is a piece of a program. It is composed of a set of C functions, some of which are mandatory. For example, for a printer device, some typical functions only called by the kernel, may be:

- **lp_init()**: Initializes the driver and is called only at boot time.
- **lp_open()**: Opens a connection with the device.
- **lp_read()**: Reads from the device.
- **lp_write()**: Writes to the device.
- **lp_ioctl()**: Performs device configuration operations.
- **lp_release()**: Interrupts connection with device.
- **lp_irqhandler()**: Specific functions called by the device to handle interrupts.

Some additional functions are available for particular applications, like ***_lseek()**, ***_readdir()**, ***_select()** and ***_mmap()**. You may find more information about them in Michael Johnson's *Hacker's Guide* (see Resources).

Do I Really Need to Write a Driver?

There are several reasons for writing our own device driver:

- To solve concurrency problems when two or more processes try to access a device at the same time.
- To use hardware interrupts: as the kernel runs in protected mode, the user cannot manage interrupts directly from a program.
- To handle other unusual applications, such as managing a virtual device (a RAM disk or a device simulator).
- To obtain satisfaction as a programmer: writing a driver increases personal motivation as well as control over the computer.

- To learn about the internal parts of the system.

Conversely, there are also several reasons for not writing our own driver:

- It requires a good deal of mental preparation.
- It requires low-level programming, i.e., direct management of ports and interrupt handlers.
- In the debug process, the kernel hangs easily, and it is not possible to use debuggers or C library functions such as **printf**.

In order to understand the following explanation, you must know the C programming language, the basic I/O procedures, a minimum about the internal architecture of a PC and have some experience in the development of software applications for Unix systems.

Finally, we must add that writing our own device driver is only necessary when the device manufacturer does not supply a driver for our OS or when we wish to add extra functionality to the one we have.

An Actual Example of a Driver

The first question we answer is: why use Linux as an example of how to write a driver? The answer is twofold: all the source files are available in Linux, and I have a working example at my lab in UPM-DISAM, Spain.

However, both the directory structure and the driver interface with the kernel are OS dependent. Indeed small changes may appear from one version or release to the next. For example, several things changed from Linux 1.2.x to Linux 2.0.x, such as the prototypes of the driver functions, the kernel configuration method and the Makefiles for kernel compilation.

The device we have selected for our explanation is the MRV-4 Mobile Robot from the U.S. company Denning-Brach International Robotics. Although the robot uses a PC with a specific board for hardware interfacing (a motor/sonar card), the company does not supply a driver for Linux. Nevertheless, all the source files of the software, which control the robot through the motor/sonar card, are available in C language for MS-DOS. The solution is to write a driver for Linux. In the example, we use kernel release 2.0.24, although it will also work in later versions with few modifications.

The mobile platform is composed of a set of wheels coupled with two motors (the drive and the steer), a set of 24 sonars which act as proximity sensors for obstacle detection and a set of bumpers which detect collisions. We need to implement a driver with, at least, the following services (**init**, **open** and **release** are mandatory):

- **write**: to send linear and angular velocity commands
- **read**: to read sonar measures and encoder values

- **three interrupt handlers**: to store sonar measures when a sonar echo is received, to implement an emergency stop when a bumper detects a collision and to stop the steer motor when the wheels are located at 0 (zero) degrees and a *go to home* flag is active
- **ioctl commands**: *go to home* which sends a constant angular velocity to the wheels and activates the *go to home* flag; and configuration of motors and sonars

The *go to home* service allows the user to stop the wheels at an initial position which is always the same (0 degrees). The incoming values from sonars and encoders, as well as the velocity commands, might be part of the main loop of the control program of the robot.

Returning to the initial scheme (Figure 1), the device is the MRV-4 robot, the hardware interface is the motor/sonar card, the source file of the driver will be mrv4.c, the new kernel we will generate will be vmlinuz, the user program for kernel testing will be **mrv4test.c** and the device will be /dev/mrv4 (see Figure 6).

Figure 6. mrv4hard MRV-4 Scheme

General Programming Considerations

To build a driver, these are the steps to follow:

1. Program the driver source files, giving special attention to the kernel interface.
2. Integrate the driver into the kernel, including in the kernel source calls to the driver functions.
3. Configure and compile the new kernel.
4. Test the driver, writing a user program.

The directory structure of the Linux source files can be described as follows: the /usr/src contains subdirectories such as /xview and /linux. Inside the /linux directory, the different parts of the kernel are classified into subdirectories: init, kernel, ipc, drivers, etc. The directory /usr/src/linux/drivers/ contains the driver sources, classified into categories such as block, char, net, etc.

Another interesting directory is /usr/include, where the main header files, such as stdio.h, are located. It contains two special subdirectories:

- /usr/include/system/, which includes system header files, such as types.h
- /usr/include/linux/, which includes the Linux kernel headers such as lp.h, serial.h, mem.h and mrv4.h.

The first task when programming the source files of a driver is to select a name to identify it uniquely, such as hd, sd, fd, lp, etc. In our case we decided to use mrv4. Our driver is going to be a character driver, so we will write the source into the file /usr/src/linux/drivers/char/mrv4.c, and its header into /usr/include/linux/mrv4.h.

The second task is to implement the driver I/O functions. In our case, **mrv4_open()**, **mrv4_read()**, **mrv4_write()**, **mrv4_ioctl()** and **mrv4_release()**.

Special care must be taken when programming the driver because of the following limitations:

- Standard library functions are not available.
- Some floating-point operations are not available.
- Stack size is limited.
- It is not possible to wait for events, because the kernel, and so all the processes, are stopped.

The OS functions supported at kernel level are, of course, only those functions programmed inside it:

- **kmalloc()**, **kfree()**: memory management
- **cli()**, **sti()**: enable/disable interrupts
- **add_timer()**, **init_timer()**, **del_timer()**: timing management
- **request_irq()**, **free_irq()**: irq management
- **inb_p()**, **outb_p()**: port management
- **memcpy_*fs()**: data management
- **printk()**: input/output
- **register_*dev()**, **unregister_*dev()**: device management
- ***sleep_on()**, **wake_up*()**: process management

Detailed information on these functions is given in Johnson's *Guide* (see Resources) or even inside the kernel source files.

Low-Level Programming

Access to the hardware interface (the card) is provided through low-memory addressing. The I/O registers of the card, where we can read/write information, are physically connected to memory addresses of the PC (i.e., ports). For instance, the motor/sonar card of the MRV-4 mobile robot is associated with the address **0x1b0**. Sixteen registers are used in this card, so the port map includes addresses from **0x1b0** to **0x1be**. A typical list of port addressing is shown in Table 1.:

Table 1. Typical Port Addresses
(/files/linuxjournal.com/linuxjournal/articles/024/2476/2476t1.html)

A free address region must be found to allocate the ports for the new card. In Table 1, addresses from **1b0** to **1be** were free. The source code example, foo.c, is available on the SSC FTP site (see end of article) and includes a call to a system function that allows us to see the previous table of addresses. Finally, access to the ports is granted via the functions **inb_p** and **outb_p**.

Interrupts are the other main topic when talking about low-level programming and hardware control. Interrupt handling versus polling has the main advantage that hardware is usually slow. We cannot stop all processes in a computer until a printer finishes a job. Instead, we can continue with normal work until the printer finishes, then send an interrupt signal that is handled by a specific function.

Continuing with our example, we need three handlers, one for each of the hardware interrupts that the card can generate: sonars handler (at irq **0x0a**), home handler (at irq **0x0b**) and bumper handler (at irq **0x0c**). As example of what the source code must do, we show the structure of the **sonar_irq_hdlr** function. Each time an echo from a sonar is received, it must:

1. Disable hardware interrupts.
2. Read sonar value from its port and store it in a driver internal variable.
3. Enable interrupts again.

If a user program wants to read the incoming data from the sonars, it must perform a **mrv4_read** operation, which returns the data stored in the internal variables of the driver.

Implementation of Driver Functions

Although we will explain the guidelines to implement each of the driver functions, when programming your own driver it is a good idea to use the driver most similar to yours as an example. In our case, the models for mrv4.c and mrv4.h are lp.c and lp.h, respectively.

The file mrv4.c includes the initialisation and I/O functions. The initialisation function **mrv4_init** must follow these steps (see guidelines in file foo.c):

1. Check in the device.
2. Get a free region for port addressing.
3. Test if hardware is present.
4. Test if irq numbers are free.
5. Initialise driver internal variables.
6. Return an OK status.

If an error is detected in any of these steps, it must undo all previous operations and return an error status. To implement the I/O functions, the following structure (or similar) must be defined and initialized in mrv4.c:

```
static struct   file_operations mrv4_fops = {
NULL,   /* mrv4_lseek   */
mrv4_read,      /* mrv4_read    */
mrv4_write,     /* mrv4_write   */
NULL,   /* mrv4_readdir */
NULL,   /* mrv4_select  */
mrv4_ioctl,     /* mrv4_ioctl   */
NULL,   /* mrv4_mmap    */
mrv4_open,      /* mrv4_open    */
mrv4_release    /* mrv4_release */
};
```

Pointers to all existent I/O functions must be set in this structure. Then, the I/O function code can be implemented, following the guidelines shown in the sidebar.

The available commands are defined in the file mrv4.h (see guidelines in file foo.h also available on the FTP site):

```
#define MRV4_MAGIC, 0x07
#define MRV4_RESET      _IO(MRV4_MAGIC, 0x01
#define MRV4_GOTOHOME   _IO(MRV4_MAGIC, 0x02
#define MRV4_RESETHOME  _IO(MRV4_MAGIC, 0x03
#define MRV4_JOYSTICK   _IOW(MRV4_MAGIC, 0x04,
        unsigned int
#define MRV4_PREPMOVE   _IOW(MRV4_MAGIC, 0x05,
        unsigned int
#define MRV4_INITODOM   _IO(MRV4_MAGIC, 0x06
#define MRV4_SONTOFIRE  _IOW(MRV4_MAGIC, 0x07,
unsigned int
```

The **_IO** macro is used for commands without arguments. The **_IOW** is used for commands with input arguments. In this case, the macro needs the argument type, for example a pointer might be of type **unsigned int**. The magic number must be chosen by the programmer. Try to select one not reserved by the system (see other header files at

/usr/include/linux). Constants are defined in the file /usr/include/linux/mrv4.h, which must be included by both the driver (mrv4.c) and the user programs. In general, the mrv4.h file can include:

- Constants and macros definitions
- ioctl commands
- Port names
- Type definitions
- Data structures to be exchanged between the driver and the user
- mrv4_init() function prototype

Driver Integration in the Kernel
The task of integrating the driver into the kernel includes several steps:

- Insert kernel calls to the new driver.
- Add the driver to the list of drivers.
- Modify compilation scripts.
- Re-compile the driver.

The insertion of the OS call to mrv4_init() is done in the /usr/src/linux/kernel/mem.c file. The other driver function calls (open, read, write, ioctl, release, etc.) are user transparent. They are carried out through the file_operations structure. A driver major number must be added to the list located at /usr/include/linux/major.h. Search for a free driver number; for example, if number 62 is free, you must add one or both of the following lines to the file, depending on the Linux release:

```
/dev/mrv4 62
#define MRV4_MAJOR 62
```

Each device is referenced by one major and one minor number. The major number represents the number of the driver. The minor number distinguishes between several devices which are controlled by the same device (e.g., several hard disks controlled by the same IDE driver: hd0, hd1, hd2).

The next step is to create a logical device to access the driver. You must use the command **mknod** in this way:

```
mknod -m og+rw /dev/mrv4 c 62 0
```

where **62** is the major, **0** the minor (only one physical device) and **c** indicates a character device. Set the permissions as necessary, although you can modify them later with the command **chmod**. For example, enable **rw** if you want to allow all users to access the device:

```
crw-rw-rw-2 bin bin 62, 0 Mar 12 1997 /dev/mrv4
```

Driver Compilation and Testing

To allow driver compilation within the kernel, the following lines must be added to the script file /usr/src/linux/arch/i386/config.in:

```
comment 'MRV 4'
bool 'MRV 4 card support' CONFIG_MRV4
```

and the following lines to /usr/src/linux/drivers/char/Makefile:

```
ifdef CONFIG_MRV4
  L_OBJS += mrv4.o
  endif
```

It is recommended that the driver be compiled alone, before linking the kernel. This method will save time testing syntax errors:

```
cd /usr/src/linux/drivers/char
gcc -c mrv4.c -Wall -D__KERNEL__
```

And when all is well, delete the object file:

```
rm -f mrv4.o
```

Next, configure the kernel by typing:

```
cd /usr/src/linux
make config
```

Answer **yes** when the script asks you about installing the MRV-4 driver (this sets the constant **CONFIG_MRV4**). Finally, insert an empty floppy disk and re-build the kernel by typing the following commands:

```
make zdisk    # generate a bootable
# floppy disk
dev -R /dev/fd0 1   # disable writes to<\n>
# floppy
```

Once you are sure that the kernel works, you can overwrite the file vmlinuz with the new kernel. To test the new kernel, restart the system (type **reboot**) and ... good luck! There are no debuggers available.
If the kernel seems to work, you might test the driver by writing one or more user programs, i.e., mrv4test.c which call the driver functions:

```
fid = open("/dev/mrv4", ...);
read(fid, ...);
write(fid, ...);
ioctl(fid, ...);
close(fid);
```

How to Obtain Additional Information
You can obtain privileged documentation at sunsite.unc.edu (see Resources). But of course, you will never be able to write your own driver using only the general guidelines of this article. To facilitate this task, we supply the source files for a dummy driver for Linux 2.0.24, which is a model for character driver development. It simulates the equation **y = a x** and includes an example of interrupt management (which does not work since it is not associated with any hardware). Its name is foo, since Linux already has a driver called dummy. These files are:

- **README**: summary of instructions to install it
- **foo.c**: driver source file
- **foo.h**: driver header file
- **footest.c**: program for driver testing

You can obtain these files via anonymous FTP at ftp://ftp.linuxjournal.com/pub/lj/listings/issue48/2476.tgz.

Guidelines (/files/linuxjournal.com/linuxjournal/articles/024/2476/2476s1.html)

Resources (/files/linuxjournal.com/linuxjournal/articles/024/2476/2476s2.html)
Fernando Matía is an Associate Professor at the Universidad Politecnica de Madrid (UPM). He was born in Madrid, Spain, in 1966. He became an Industrial Engineer at UPM in 1990 and received his Ph.D. degree at UPM in 1994 in the area of Control

Engineering. He works at the Systems and Automatic Control Engineering Division (DISAM). His main activities are Intelligent Control, Fuzzy Control, Robotics and Computer Sciences. He can be reached at matia@disam.upm.es.

You May Like

What Does It Take to Make a Kernel? (/content/what-does-it-take-make-kernel-0)

Petros Koutoupis (/users/petros-koutoupis)

(/content/what-does-it-take-make-kernel-0)

Oracle Linux on Btrfs for the Raspberry Pi (/content/oracle-linux-btrfs-raspberry-pi)

Charles Fisher (/users/charles-fisher)

(/content/oracle-linux-btrfs-raspberry-pi)

When Choosing Your Commercial Linux, Choose Wisely! (/content/when-choosing-your-commercial-linux-choose-wisely)

(/content/when-choosing-your-commercial-linux-choose-wisely)

Vince Calandra (/users/vince-calandra)

Arduino from the Command Line: Break Free from the GUI with Git and Vim! (/content/arduino-command-line-break-free-gui-git-and-vim)

(/content/arduino-command-line-break-free-gui-git-and-vim)

Matthew Hoskins (/users/matthew-hoskins)

Connect With Us

(https://youtube.com/linuxjournalonline) (https://www.facebook.com/linuxjournal/) (https://twitter.com/linuxjournal)

Linux Journal, representing 25+ years of publication, is the original magazine of the global Open Source community.