Fail Safe Port Allocation for Linux Device Drivers

Abstract

Writing a device driver is a challenging and an adventurous job. Once the device is registered in the driver's init_module routine, the resources for the device should be allocated. One of the main resources for the device is I/O port. The dynamically linked drivers, the developer should be careful to allocate unused range of port addresses for their device. First the driver should probe whether the range of ports are in use or free, then the ports should be requested for the device. When the module is removed from the kernel, the ports must be freed. This article discusses the intricacies of fail safe port allocation for the Linux device

by Dr. B. Thangaraju
<balasubramanian.thangaraju(at)wipro.com>

About the author:
Dr. B. Thangaraju received a Ph.D in Physics from Bharathidasan University, Tamil Nadu and worked as a Research Associate for five years in Indian Institute of Science, India. He has done research in the areas of Transparent and Conducting Oxide (TCO) thin films, Spray Pyrolysis, Photo Acoustic Techniques and p- to n- transition of Chalcogenide Glasses. He has published 10 research papers in renowned international journals. He has also presented his research findings in more than seven international and national conferences.

He is presently working as a Manager in Talent Transformation, Wipro Technologies, India. His current areas of research, study and knowledge dissemination are Linux Kernel, Device Drivers and Real Time Linux.

Content:

Introduction

The major concern of a device driver developer is allocating resources for the device. The resources are I/O ports, memory and IRQ lines. This article attempts to explain the fundamentals of I/O subsystems and the importance of allocating resources, mainly the resources dealing with the I/O ports. It also clarifies how to probe, request and free the port addresses for the devices.

The basic hardware elements, such as ports, buses and device controllers, accommodate a wide variety of I/O devices. The device drivers present a uniform device access interface to the I/O subsystem much as the system calls provide a standard interface between the application and the operating system. There are many kinds of devices, which are attached to the computer, for example, storage devices like disks, tapes, CD-ROM and floppies, human interface devices like keyboard, mouse and screen, transmission devices like network cards and modems. Despite the large number of the various types of I/O devices, we need to only understand a few basic concepts to how the devices are attached and how the software can control the hardware.

Fundamental Concept

A device consists of two parts, one is electronic part, which is called a device controller and another one is a mechanical part. The controller is attached to the system through the system bus. Typically, set of port addresses (non-conflicting) are attached to each controller. I/O ports comprises of four sets of registers such as status, control, data-in and data-out. The status register has bits that can be read by the host, which indicate whether the current command is completed or a byte is ready to be read or written or for any error notification. The control register is written by the host to start a command or to change the mode of a device. The data-in register is for getting input and data-out register is for sending output to the system.

So, the basic interface between the processor and a device is a set of control and status registers. When the processor is executing a program and encounters an instruction relating to the device, it executes the instruction by issuing a command to the appropriate device. The controller performs the requested action and then sets the appropriate bits in the status register and waits. It is the responsibility of the processor to check the status of the device periodically until it finds that the operation is complete. For example, the parallel port driver (used by printer) normally polls the printer to see if the printer is ready to accept output, and if the printer is not ready, the driver will sleep for a while (processor can do some useful work), and try again and again until the printer is ready. This polling mechanism will improve the system performance, otherwise the system is unnecessarily waiting for the device without doing any useful job.

The registers have a well-defined address in the I/O space. Generally, these addresses are assigned at boot time, using a set of parameters specified in a configuration file. A range of addresses might be allocated for each device, if the device is attached statically. This means the kernel contains the device drivers for the existing devices, the allocated I/O port ranges for the device can be stored in the **proc** directory. You can check the existing port address ranges for the devices which your system is using currently from **$cat /proc/ioports**. The first column of this output shows the ranges of the port and the second column is the device that owns the ports. Some of the operating systems have the features of loading the device driver module dynamically when the system is running. So, any new device can be attached to the system, while the system is running and can be controlled / accessed by the dynamically attached device driver module for the newly attached device.

The device driver concept is quite abstract and it is the lowest level of the software that runs on a computer, as it is directly bound to the hardware features of the device. Each device driver manages a single type of device. The type may be character, block or network. If an application requests the device, the kernel contacts the appropriate device driver. The driver then issues the command to the particular device. The device driver is a collection of functions: it has many entry points like open, close, read, write, ioctl, llseek etc. When you insert your module, the init_module ( ) function is called and when the module is removed, the cleanup_module ( ) function is called. The device is registered in a device driver in the init_module ( ) routine.

When a device is registered in the init_module ( ), then the resources for the device like I/O ports, memory and IRQ lines are allocated in the function itself, which are needed for the driver for correct device operation. If you allocate any wrong memory address for the device, the kernel shows the error message **segmentation fault**. But in the case of I/O ports, the kernel will not give any error messages like **wrong I/O port** but assigning already used ports, which belongs to existing devices will crash your system. When you remove the module, the device should be unregistered, that is the major number will be released and the resources will be freed in the cleanup_module ( ) function.

The most frequent job of the device driver is reading and writing I/O ports. So, your driver should be pretty sure that the port addresses are used by the device is exclusive. Any other devices should not use the range of addresses. To ensure this first the driver should probe whether the port address is already in use or not: Once the driver finds the addresses are not in use, it can request the kernel to allocate the range of addresses to its device.

Fail Safe Port Allocation

Now we will see how to implement the resource allocation and free the resources with the kernel functions. The practical approach is experimented with the Linux 2.4 kernel. Hereafter the complete implementation is applicable to only Linux operating system and some extent the other UNIX variants.

First, to probe the range of ports available or not for a device is done by

```
int check_region (unsigned long start, unsigned long len);
```

the function returns zero if the range of port addresses is available or less than zero or negative error code (-EBUSY or -EINVAL) if it is already in use. The function accepts two arguments: **start** is the beginning of the contiguous region (or range of I/O ports) and the **len** is the number of ports in the region.

Once the port is available, it should be allocated for the device, which can be done by the request_region function.

```
struct resource *request_region (unsigned long start, unsigned long len, char *name);
```

The first two arguments are the same as we have seen earlier, the character pointer variable **name** is the name of the device, which the port address is being allocated. The function returns the type of pointer to the struct resource. Resource structure is used to describe the resource ranges, which is declared in <linux/ioport.h>. The structure contains the following format:

```
struct resource {
        const char *name;
        unsigned long start, end;
        unsigned long flags;
        struct resource *parent, *sibiling, *child;
};
```

When the module is removed from the kernel, the port should be released for other devices to use for this we have to use release_region ( ) function in the cleanup_module ( ). The syntax of the function is

void release_region (unsigned long start, unsigned long len);

The explanation of the two arguments are the same as before. The above three functions are actually macros, which are declared in <linux/ioport.h>.

Example Driver Code for Device Port Allocation

The following program explains the allocation and de-allocation for ports for your dynamically loaded device.

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

struct file_operations fops;
unsigned long start, len;

int init_module (void)
{
 int status;
 start = 0xff90;
 len   = 0x90;

 register_chrdev(254,"your_device",&fops);

 status =  check_region (start, len);
 if (status == 0) {
     printk ("The ports are available in that range.\n");
     request_region(start,len,"your_device");
 } else {
     printk ("The ports are already in use. Try other range.\n");
     return (status);
 }
 return 0;
}

void cleanup_module (void)
{
 release_region(start, len);
 printk (" ports are freed successfully\n");
 unregister_chrdev(254,"your_device");}
 printk (" your device is unregistered\n");
}
```

To avoid confusion, the error checking and dynamic major number allocation are avoided in this example code. Once the port is allocated successfully, we can check in the proc directory:
$cat /proc/ioports

Kernel I/O Port Function Options for Driver

Linux supports a variety of functions dependent on the size of the ports, to read from and write to I/O ports. Ports can be 8, 16 or 32 bit wide. The Linux kernel headers <asm/io.h> define the inline functions to access I/O ports. For reading (inx) and writing (outx) 8 bit, 16 bit and 32 bit ports, the following functions are in use:

```
__u8 inb (unsigned int port);
void outb (__u8 data, unsigned int port);

__u16 inw (unsigned int port);
```

```
void outw(__u16 data, unsigned int port);

__u32 inl (unsigned int prot);
void outl (__u32 data, unsigned int port);
```

For string versions that allow you to transfer more than one datum at a time efficiently by using the following functions.

```

void insb(unsigned int port, void *addr, unsigned long count);
void outsb(unsigned int port, void *addr, unsigned long count);
```

addr is the location in memory to transfer to or from and count is the number of units to transfer. Data is read from or written to the single port "port".

```

void insw(unsigned int port, void *addr, unsigned long count);
void outsw(unsigned int port, void *addr, unsigned long count);
```

read or write 16 bit values to a single 16 bit port.

```

void insl(unsigned int port, void *addr, unsigned long count);
void outsl(unsigned int port, void *addr, unsigned long count);
```

read or write 32 bit values to a single 32 bit port.

Acknowledgment

Author is very grateful to **Mr. Jayasurya V**, Manager, Talent Transformation, Wipro Technologies, India, for his critical reading of the manuscript.

References

- Linux Device Drivers (2nd Edition), by Alessandro Rubini and Jonathan Corbet. The book is available from o'reilly: http://linux.oreilly.com
- Linux Kernel 2.4 Internals: http://tldp.org/LDP/lki/index.html
- Linux Kernel Module Programming Guide: http://tldp.org/LDP/lkmpg/mpg.html
- The Linux Kernel (older guide, 1998): http://tldp.org/LDP/tlk/tlk.html

Webpages maintained by the LinuxFocus Editor team © Dr. B. Thangaraju, FDL LinuxFocus.org	Translation information: en --> -- : Dr. B. Thangaraju <balasubramanian.thangaraju(at)wipro.com>