LINUX GAZETTE
...making Linux just a little more fun!

Risk-Free Resource Allocation for I/O Memory-Mapped Device Drivers

By Dr B Thangaraju

Abstract

A device driver is an entry point to access a device. Developing a device driver is not as simple a task as writing application programs. Since any dynamically-loaded driver module is attached to the existing kernel, any error in the driver will crash the entire system. Resource allocation for a device is one of the main concerns for device driver developers. The device resources are I/O memory, IRQs and ports. This article presents a risk-free way of allocating resource for an I/O memory mapped device for a dynamically loaded Linux device driver, and is written so that less experienced Linux users can follow along.

Introduction

In the rapidly developing IT field, new devices are constantly being developed and we see an increasingly wide variety of Input and Output devices. The I/O subsystem allows a process to communicate with peripheral devices such as disks, floppies, CD-ROMs, terminals, printers and networks. Kernel modules that control devices are known as device drivers. The I/O subsystem handles the movement of data between memory and peripheral devices. The type of the devices can be classified into character and block depending on the way the system accesses the device. In general, the character devices like keyboard, mouse, console and modem are accessed as a stream of bytes. However, the block devices like Hard disk, floppy and CD-ROM move blocks of data to and from the system.

The kernel interacts with these devices through device drivers. A device driver is a collection of functions used to access any particular device. One of the important features of Linux is that the device driver module can be inserted dynamically into the existing kernel. Then the driver module will become part of the kernel and can access the kernel functions. In the same way, the loaded driver can be removed dynamically. If the driver is not explicitly removed, it will be persistent in the system until we reboot the machine.

The most frequent job of any driver is transferring data between the computer and its external environment. The external environment consists of a variety of external devices, including secondary memory devices, communications equipment and terminals. Three techniques are possible for I/O operations: programmed I/O, Interrupt - driver I/O and Direct Memory Access (DMA). The programmed I/O devices pass the data from system to device or vice versa in two different ways: I/O port and I/O memory mapped. This article explains the basic concept of I/O memory mapped devices and the macros used by the device driver for allocating I/O memory regions for the device and expounds the concept with well tested device driver code. Since the driver module is part of the kernel, any attempt to allocate existing address to your device will crash the system. So the sample driver will first probe whether the address range is free or not, if it is already in use by other device it will return immediately with an error, otherwise it will allocate the given address range to your device.

Basics of I/O memory mapped device

Device drivers are extremely device dependent. The driver framework should take responsibility of how the CPU interacts with the device. PIO and DMA are the two ways of moving data between the kernel and the device. PIO requires the CPU to move data to or from the device as each byte is ready, by responding to an interrupt or polling. For DMA devices, the kernel gives the source address, the destination address and the size of the data in memory. The device can transfer the data without CPU intervention, and when the data is moved it will send an interrupt to notify the kernel of the completion. Typically, slow devices like modem and line printers are PIO devices, while disks and graphics terminals are DMA devices.

For PIO devices, there are two ways to pass data from the device to system memory. Which way a system uses depends on its architecture. For instance, Intel x86 architectures supports I/O port, and Motorola 680x0 maintains memory mapped device I/O. Moreover, most of the ISA (Industry Standard Architecture) devices belong to the I/O support allocation and PCI (Peripheral Component Interconnect) devices uphold I/O memory mapped allocation. Several parameters that a driver must know are, for example, the hardware's actual I/O addresses or memory range. Sometimes you need to pass parameters to a driver to help it in finding its own device or to enable/disable specific features. In my previous article in Linux Focus http://linuxfocus.org/English/November2002/article264.shtml, I explained the fundamentals of device controllers and intricacies of the fail safe port allocation for Linux device drivers From the driver developer perspective, the allocation of I/O memory for a device has some similarity with allocation of I/O ports because both are based on similar internal mechanisms. So it is redundant to explain again the basics of device controller and the functions of status and control registers for transferring data from or to device to system memory.

Macros used for I/O memory address allocation

To probe whether the address range is already in use or not, use the following macro in driver.

int check_mem_region (unsigned long start, unsigned long length);

Here, the first argument **start** is the starting address of the I/O memory and **length** is the size of the address range. The function returns zero if the address range is available otherwise returns less than zero. To register the given I/O memory regions, the macro is

void request_mem_region (unsigned long start, unsigned long length, char *device_name);

The string argument **char *device_name** is the name of device, which will own the I/O memory regions from start address to length size. Before the device is unregistered, the allocated I/O memory regions should be released for other devices.

void release_mem_region (unsigned long start, unsigned long length);

The above function will de-allocate the I/O memory regions.

Example Driver Code for I/O memory region allocation

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

static int Major, result;
struct file_operations fops;

unsigned long start = 0, length = 0;

MODULE_PARM (start, "l");
MODULE_PARM (length, "l");

int Wipro_init (void) {
   Major = register_chrdev (0, "Wipro_device", &fops);
   if (Major < 0)
   {
      printk (" Major number allocation is failed \n");
      return (Major);
   }
   printk (" The Major number of the device is %d \n", Major);

   result = check_mem_region (start, length);
   if (result < 0)
      {
         printk ("Allocation for I/O memory range is failed: Try other range\n");
         return (result);
      }

      request_mem_region (start, length, "Wipro_device");
      return 0;
}

void Wipro_cleanup (void) {
    release_mem_region (start, length);
    printk (" The I/O memory region is released successfully \n");

    unregister_chrdev (Major, "Wipro_device");
    printk (" The Major number is released successfully \n");
}

module_init (Wipro_init);
module_exit (Wipro_cleanup);
```

The above program is saved as io_mem.c. First four lines are the headers, which are included to access kernel macros and functions. Next is the variable and file_operations structure declaration. The macro MODULE_PARM is the driver modules parameter for assigning value of any variable during the module loading. It will accept two arguments, first one is the variable name and the second one is data type of the variable. In this code, "**l**" means long int.

In Wipro_init and Wipro_cleanup functions are explicit initialization and cleanup for this driver module. The modern mechanism recommends this approach for marking init_module and cleanup_module. The Wipro_init function first registers the Wipro_device and allocates major number dynamically. Then it will probe the given address, **if the address is already in use, the function will return an error, otherwise it will allocate the address range for the device**. The Wipro_cleanup function deallocate the I/O memory region before unregistering the device name and major number.

The file is compiled with 2.4 kernel and has been created io_mem.o object file. The following **device** and **iomem** file contains part of the existing data in my computer shown below.

```
$cat /proc/devices
Character devices:
1 mem
2 pty
...
180 usb

$cat /proc/iomem
00000000-0009fbff : System RAM
...
e0000000-e3ffffff : Silicon Integrated Systems [SiS] 620 Host
ffff0000-ffffffff : reserved
```

The module is loaded by a command, **$insmod ./io_mem.o start=0xeeee0000 length=0xeeee**. After loading successfully, it is evident that the device is registered with the existing devices list with major number 254 and the given memory range is allocated and shown in devices and iomem file respectively as shown below.

```
$cat /proc/devices
Character devices:
1 mem
2 pty
...
180 usb
254 Wipro_device

$cat /proc/iomem
00000000-0009fbff : System RAM
...
e0000000-e3ffffff : Silicon Integrated Systems [SiS] 620 Host
eeee0000-eeeeeeed : Wipro_device
ffff0000-ffffffff : reserved
```

Conclusion

We discussed the importance of the risk-free resource allocation for I/O memory mapped devices for Linux device drivers. We examined the basics of I/O memory mapped devices, the macros for I/O memory address allocation. We explained the practical approach of how to allocate resource for I/O memory mapped devices with the well tested device driver code. We verified the code is explained and the device register and memory range address allocation.

Acknowledgment

I would like to acknowledge **Mr.V.Jayasurya and Dr. Sanjay Gupta** ,Talent Transformation, Wipro Technologies, India.

References

1. Linux Device Drivers (2nd Edition), by Alessandro Rubini and Jonathan Corbet. The book is available from O'Reilly : http://linux.oreilly.com/

Dr B Thangaraju

E-mail:balasubramanian.thangaraju@wipro.com
Updated: 2002-09-27

Dr B Thangaraju received a Ph.D in Physics and worked as a Research Associate for five years in Indian Institute of Science, India. He is presently working as a Manager in Talent Transformation, Wipro Technologies, India. He has published many research papers in renowned international journals. His current areas of research, study and knowledge dissemination are Linux Kernel, Device Drivers and Real-Time Linux.