Linux Character Device Drivers I

Important Notice : Courseware - Legal

This courseware is both the product of the author and of freely available opensource and/ or public domain materials. Wherever external material has been shown, it's source and ownership have been clearly attributed. We acknowledge all copyrights and trademarks of the respective owners.

The contents of the **courseware PDFs are considered proprietary** and thus cannot be copied or reproduced in any form whatsoever without the explicit written consent of the author.

Only the programs - **source code** and binaries (where applicable) - that form part of this courseware, and that are made available to the participant, are released under the terms of the permissive **MIT license** [1].
Under the terms of the MIT License, you can certainly use the source code provided here; you must just attribute the original source (author of this courseware and/or other copyright/trademark holders).

VERY IMPORTANT :: Before using this source(s) in your project(s), you *MUST* check with your organization's legal staff that it is appropriate to do so.

The courseware PDFs are *not* under the MIT License, they are to be kept confidential, non-distributable without consent, for your private internal use only.

The duration, contents, content matter, programs, etc. contained in this courseware and companion participant VM are subject to change at any point in time without prior notice to individual participants.

Care has been taken in the preparation of this material, but there is no warranty, expressed or implied of any kind, and we can assume no responsibility for any errors or omisions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Device Drivers : an article by Michael K. Johnson

If you choose to write a device driver, you must take everything written here as a guide, and no more. I cannot guarantee that this chapter will be free of errors, and I cannot guarantee that you will not damage your computer, even if you follow these instructions exactly. It is highly unlikely that you will damage it, but I cannot guarantee against it. There is only one ``infallible'' direction I can give you: Back up! Back up before you test your new device driver, or you may regret it later.

What is a Device Driver?

Making hardware work is tedious. To write to a hard disk, for example, requires that you write magic numbers in magic places, wait for the hard drive to say that it is ready to receive data, and then feed it the data it wants, very carefully. To write to a floppy disk is even harder, and requires that the program supervise the floppy disk drive almost constantly while it is running.

Instead of putting code in each application you write to control each device, you *share* the code between applications. To make sure that that code is not compromised, you protect it from users and normal programs that use it. If you do it right, you will be able to add and remove devices from your system without changing your applications at all. Furthermore, you need to be able to load your program into memory and run it, which the operating system also does. So an operating system is essentially a priviledged, general, sharable library of low-level hardware and memory and process control functions and routines.

> Operating systems use an abstract method of performing I/O on devices; both UNIX and Linux abstract devices as files - device special files

All versions of Unix have an *abstract way* of reading and writing devices. By making the devices act as much as possible like regular files, the same calls (read(), write(), etc.) can be used for devices and files. Within the kernel, there are a set of functions, registered with the filesystem, which are called to handle requests to do I/O on ``*device special files,*'' which are those which represent devices. (See mknod(1,2) for an explanation of how to make these files.)

Block and Character Devices

Character devices:
-do not have a physically-addressable media
-data is accessed as a stream of bytes (eg. the console and the serial ports)
-cannot be seeked to
-does not have a size

Block devices:
-capability to mount a filesystem.
-access data in blocks that are a power of 2 (usually a few kilobytes);
-they are random-access
-are seekable and have a size
-always accessed through the page cache

Both are accessed via special device files in the filesystem (/dev/xxx)

A block device is one in which the data that moves to and from it occurs in blocks (such as disk sectors) and supports attributes such as buffering and random access behavior (is not required to read blocks sequentially, but can access any block at any time). Block devices include hard drives, CD-ROMs, and RAM disks.

This is in contrast to character devices, which differ in that they do not have a physically-addressable media. Character devices include serial ports and tape devices, in which data is streamed character by character.

Major and Minor Numbers

A device driver works on a class of device specified by the device file's major number; several device files can have the same major number but should have different minor numbers to distinguish between them.

See *Documentation/devices.txt*
[Note- moved to – on recent kernels – *Documentation/admin-guide/device.txt*]

Sidebar

Source: Linux Device Drivers, 3rd Ed (LDD3), Rubini, Corbet & Hartman, O'Reilly.

The Internal Representation of Device Numbers

Within the kernel, the `dev_t` type (defined in *<linux/types.h>*) is used to hold device numbers—both the major and minor parts. As of Version 2.6.0 of the kernel, `dev_t` is a 32-bit quantity with 12 bits set aside for the major number and 20 for the minor number.

Your code should, of course, never make any assumptions about the internal organization of device numbers; it should, instead, make use of a set of macros found in *<linux/kdev_t.h>*. To obtain the major or minor parts of a `dev_t`, use:

```
MAJOR(dev_t dev);
MINOR(dev_t dev);
```

and

```
unsigned int iminor(struct inode *inode);
unsigned int imajor(struct inode *inode);
```

If, instead, you have the major and minor numbers and need to turn them into a `dev_t`, use:

```
MKDEV(int major, int minor);
```

Note that the 2.6 kernel can accommodate a vast number of devices*, while previous kernel versions were limited to 255 major and 255 minor numbers. One assumes that the wider range will be sufficient for quite some time, but the computing field is littered with erroneous assumptions of that nature. So you should expect that the format of dev_t could change again in the future; if you write your drivers carefully, however, these changes will not be a problem.

* This gives us a possible 2^{12} (4096) major devices, each of which can have upto 2^{20} (1048576) minors!

By contrast, on 2.4 kernels and below, the major/minor pair was just 16 bits wide with 8 bits for each of them.

> The device driver code will reside in and execute in kernel space in *kernel mode* - we need to ensure that it does it's work correctly as there is no protection as there is in user-mode programming.

However, a word of warning is due here: ***Writing a device driver is writing a part of the Linux kernel***. This means that your driver runs with kernel permissions, and can do anything it wants to: write to any memory, reformat your hard drive, damage your monitor or video card, or even break your dishes, if your dishwasher is controlled by your computer. Be careful.

[The following sections are from the book "Linux Device Drivers" 3rd Ed by A Rubini, J Corbet and GK-Hartman (O'Reilly & Associates).]

Doing It in User Space

A Unix programmer who's addressing kernel issues for the first time might well be nervous about writing a module. Writing a user program that reads and writes directly to the device ports is much easier.

Indeed, there are some arguments in favor of user-space programming, and sometimes writing a so-called user-space device driver is a wise alternative to kernel hacking.

The **advantages** of user-space drivers can be summarized as follows:

- The full C library can be linked in. The driver can perform many exotic tasks without resorting to external programs (the utility programs implementing usage policies that are usually distributed along with the driver itself).
- The programmer can run a conventional debugger on the driver code without having to go through contortions to debug a running kernel.
- If a user-space driver hangs, you can simply kill it. Problems with the driver are unlikely to hang the entire system, unless the hardware being controlled is really misbehaving.
- User memory is swappable, unlike kernel memory. An infrequently used device with a huge driver won't occupy RAM that other programs could be using, except when it is actually in use.
- A well-designed driver program can still allow concurrent access to a device.
- If you must write a closed-source driver, the user-space option makes it easier for you to avoid ambiguous licensing situations and problems with changing kernel interfaces.

For example, USB drivers can be written for user space; see the (still young) libusb project at libusb.sourceforge.net and “gadgetfs” in the kernel source. Another example is the X server: it knows exactly what the hardware can do and what it can’t, and it offers the graphic resources to all X clients. Note, however, that there is a slow but steady drift toward frame-buffer-based graphics environments, where the X server acts only as a server based on a real kernel-space device driver for actual graphic manipulation.

Usually, the writer of a user-space driver implements a server process, taking over from the kernel the task of being the single agent in charge of hardware control. Client

applications can then connect to the server to perform actual communication with the device; therefore, a smart driver process can allow concurrent access to the device. This is exactly how the X server works.

But the user-space approach to device driving has a number of **drawbacks**. The most important are as follows:

- Interrupts are not available in user space. There are workarounds for this limitation on some platforms, such as the vm86 system call on the IA32 architecture.
- Direct access to memory is possible only by mmapping /dev/mem, and only a privileged user can do that.
- Access to I/O ports is available only after calling ioperm or iopl. Moreover, not all platforms support these system calls, and access to /dev/port can be too slow to be effective. Both the system calls and the device file are reserved to a privileged user.
- Response time is slower, because a context switch is required to transfer information or actions between the client and the hardware.
- Worse yet, if the driver has been swapped to disk, response time is unacceptably long. Using the mlock system call might help, but usually you'll need to lock several memory pages, because a user-space program depends on a lot of library code. mlock, too, is limited to privileged users.
- Important devices can't be handled in user space, including, but not limited to, network interfaces and block devices.

As you see, user-space drivers can't do that much after all. Interesting applications nonetheless exist: for example, support for SCSI scanner devices (implemented by the SANE package) and CD writers (implemented by cdrecord and other tools). In both cases, user-level device drivers rely on the "SCSI generic" kernel driver, which exports low-level SCSI functionality to user-space programs so they can drive their own hardware.

In order to write a user-space driver, some hardware knowledge is sufficient, and there's no need to understand the subtleties of kernel software. We won't discuss user-space drivers any further in this book, but will concentrate on kernel code instead.

One case in which working in user space might make sense is when you are beginning to deal with new and unusual hardware. This way you can learn to manage your hardware without the risk of hanging the whole system. Once you've done that, encapsulating the software in a kernel module should be a painless operation.

[OPTIONAL / FYI]

Example User-space Device Driver

Resource / Source : http://tldp.org/LDP/khg/HyperNews/get/devices/fake.html

--snip--

Example: vgalib

A good example of a user-space driver is the vgalib library. The standard read() and write() calls are really inadequate for writing a really fast graphics driver, and so instead there is a library which acts conceptually like a device driver, but runs in user space. Any processes which use it must run setuid root, because it uses the ioperm() system call. It is possible for a process that is not setuid root to write to /dev/mem if you have a group mem or kmem which is allowed write permission to /dev/mem and the process is properly setgid, but only a process running as root can execute the ioperm() call.

There are several I/O ports associated with VGA graphics. vgalib creates symbolic names for this with #define statements, and then issues the ioperm() call like this to make it possible for the process to read and write directly from and to those ports:

```
    if (ioperm(CRT_IC, 1, 1)) {
       printf("VGAlib: can't get I/O permissions \n");
       exit (-1);
    }
    ioperm(CRT_IM,  1, 1);
    ioperm(ATT_IW, 1, 1);
[…]
```

It only needs to do error checking once, because the only reason for the ioperm() call to fail is that it is not being called by the superuser, and this status is not going to change.

⚠After making this call, the process is allowed to use inb and outb machine instructions, but only on the specified ports. These instructions can be accessed without writing directly in assembly by including , but will only work if you compile with optimization on, by giving the -O? to gcc. Read <*linux/asm.h*> for details.

After arranging for port I/O, vgalib arranges for writing directly to kernel memory with the following code:

```
 /* open /dev/mem */
if ((mem_fd = open("/dev/mem", O_RDWR) ) < 0) {
    printf("VGAlib: can't open /dev/mem \n");
    exit (-1);
}

/* mmap graphics memory */
if ((graph_mem = malloc(GRAPH_SIZE + (PAGE_SIZE-1))) == NULL)
{
    printf("VGAlib: allocation error \n");
    exit (-1);
}

if ((unsigned long)graph_mem % PAGE_SIZE)
    graph_mem += PAGE_SIZE - ((unsigned long)graph_mem % PAGE_SIZE);

 /*
 void *mmap(void *addr, size_t length, int prot, int flags,
              int fd, off_t offset);
*/
graph_mem = (unsigned char *)mmap(
    (caddr_t)graph_mem,
    GRAPH_SIZE,
    PROT_READ|PROT_WRITE,
    MAP_SHARED|MAP_FIXED,
    mem_fd,
    GRAPH_BASE
);

if ((long)graph_mem < 0) {
    printf("VGAlib: mmap error \n");
    exit (-1);
}
```

It first opens /dev/mem, then allocates memory enough so that the mapping can be done on a page (4 KB) boundary, and then attempts the map. GRAPH_SIZE is the size of VGA memory, and GRAPH_BASE is the first address of VGA memory in /dev/mem. Then by writing to the address that is returned by mmap(), the process is actually writing to screen memory.

--snip--

SIDEBAR

Doing the *mmap()* on ARM, *requires* the address and length fields (params) to be page-aligned.
[Link]

Armed with this knowledge, lets try a very simple user-space device driver program.
Source: http://www.svgalib.org/jay/beginners_guide/beginners_guide.html

Easy Graphics: A Beginner's Guide to SVGAlib

(c) by Jay Link

--snip--
To use SVGAlib, you must reference it in your C program. Simply #include <vga.h> .
Here's about the easiest SVGAlib program there is:

<< Trivially enhanced to paint some colour rectangles instead of a single red pixel :-)
>>

```
#include <stdlib.h>
#include <unistd.h>
#include <vga.h>

#define MAXCOLOR      32

int main(void)
{
       int x, y, c=2;

   vga_init();
   vga_setmode(G320x200x256);
   vga_setcolor(1);

       for (x=0; x<320; x++) {
              vga_drawpixel(x, 10); // paint horizontally
              for (y=0; y<200; y++)  // paint vertically
                     vga_drawpixel(x, y);
              if (!(x % 32)) {
                     vga_setcolor(c++);
                     if (c >= (MAXCOLOR-1))
                            c=0;
              }
       }

   sleep(5);
   vga_setmode(TEXT);

   return EXIT_SUCCESS;
}
```

This will paint some colour bars ~~a single red pixel~~ on your screen. After five seconds, it will reset your console to text mode and will exit.

Note our first statement, vga_init() . This relinquishes root status and initializes the SVGAlib library. The second line, vga_setmode(G320x200x256) , sets the screen to mode 5, which is 320x200x256. That is to say, your screen becomes a grid which is 320 pixels wide, 200 pixels high, and which supports 256 colors. Alternatively, you could write vga_setmode(5). Either statement is acceptable. Our next command, vga_setcolor(4) , makes red the current color. We can choose any value from 0 to 255.

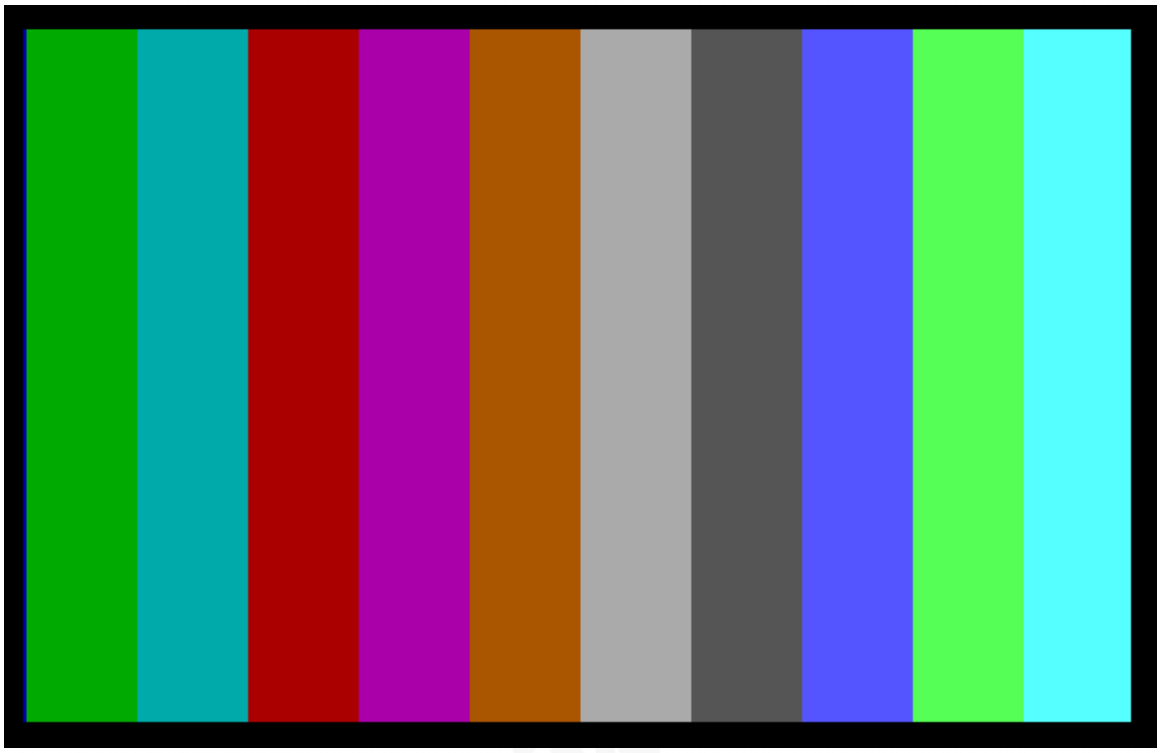
--snip--

Of course, one must install the [S]VGA library and development headers first. Compile the program with:

```
$ cc vga1.c -o vga1 -lvga -Wall
$ ldd ./vga1
        linux-gate.so.1 =>  (0x00bdf000)
        libvga.so.1 => /usr/lib/libvga.so.1 (0x00d43000)
        libc.so.6 => /lib/libc.so.6 (0x00119000)
        libx86.so.1 => /lib/libx86.so.1 (0x00819000)
        libm.so.6 => /lib/libm.so.6 (0x008a7000)
        /lib/ld-linux.so.2 (0x006df000)
$
```

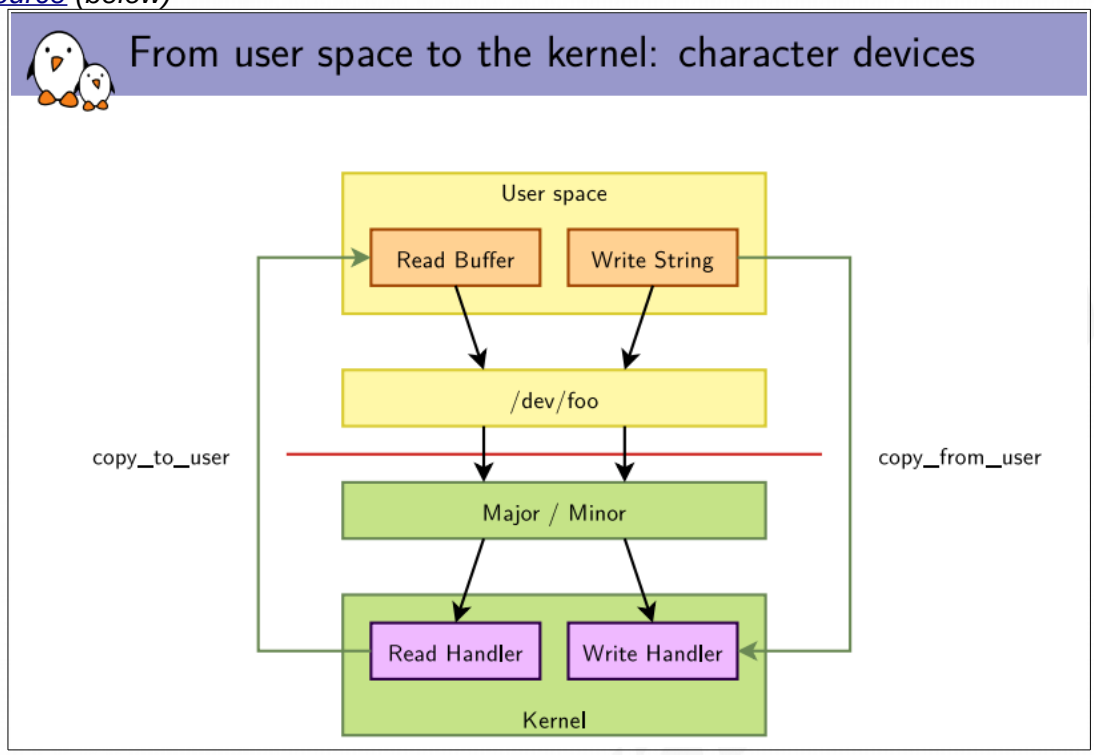
[P. T. O. →]

```
$ sudo ./vga1
...
```



Screenshot of the 'vga1' userspace driver running (within a virtual machine).

Source (below)



The kernel's “mem” driver

The mem character driver is always built into the Linux kernel. This driver comprises several common (and not so common) useful device nodes: for example, the null, zero and random devices are driven by this driver. We do not intend to cover the detailed working of the mem driver itself in this section; instead, we use this driver to study the overall *framework* of a typical Linux device driver – how it ties into userspace, the kernel (VFS) and the hardware.

<<
The 'mem' char driver has a major number of 1.
In order to understand what the memory_open() below does, we take a peek at all the char device files on the filesystem having a major number of 1 (sorted by their minor number).

```
$ ls -l /dev/* |grep '^c' | awk '$5=="1," {print $0}'|sort -k6n
crw-r-----  1 root    kmem      1,   1 May  7 13:31 /dev/mem
crw-rw-rw-  1 root    root      1,   3 May  7 13:31 /dev/null
```

```
crw-r-----  1 root    kmem      1,   4 May  7 13:31 /dev/port
crw-rw-rw-  1 root    root      1,   5 May  7 13:31 /dev/zero
crw-rw-rw-  1 root    root      1,   7 May  7 13:31 /dev/full
crw-rw-rw-  1 root    root      1,   8 May  7 13:31 /dev/random
crw-rw-rw-  1 root    root      1,   9 May  7 13:31 /dev/urandom
crw-r--r--  1 root    root      1,  11 May  7 13:31 /dev/kmsg
$
```
>>

<<
* On recent kernels, the presence or absence of */dev/kmem* depends on whether CONFIG_DEVKMEM is enabled in the kernel. For security reasons, most modern distros might have it disabled by default.
>>

The mem driver is initialized during kernel initialization, via the *chr_dev_init()* function. This function is called when the kernel first boots and is configuring itself.

<<
*Partial source listing of **drivers/char/mem.c** (on the 2.6.10 kernel).*
The code is annotated by this author as an aid to understanding.
>>

```
--- snip ---
```

<< *Define all minor-specific file_operation structures for implementing various functionalities* >>

```
static const struct file_operations __maybe_unused mem_fops = {
        .llseek         = memory_lseek,
        .read           = read_mem,
        .write          = write_mem,
        .mmap           = mmap_mem,
        .open           = open_mem,
#ifndef CONFIG_MMU
        .get_unmapped_area = get_unmapped_area_mem,
        .mmap_capabilities = memory_mmap_capabilities,
#endif
};

static const struct file_operations __maybe_unused kmem_fops = {
        .llseek         = memory_lseek,
        .read           = read_kmem,
        .write          = write_kmem,
```

```
        .mmap           = mmap_kmem,
        .open           = open_kmem,
#ifndef CONFIG_MMU
        .get_unmapped_area = get_unmapped_area_mem,
        .mmap_capabilities = memory_mmap_capabilities,
#endif
};

static const struct file_operations null_fops = {
        .llseek         = null_lseek,
        .read           = read_null,
        .write          = write_null,
        .read_iter      = read_iter_null,
        .write_iter     = write_iter_null,
        .splice_write   = splice_write_null,
};

static const struct file_operations __maybe_unused port_fops = {
        .llseek         = memory_lseek,
        .read           = read_port,
        .write          = write_port,
        .open           = open_port,
};

static const struct file_operations zero_fops = {
        .llseek         = zero_lseek,
        .write          = write_zero,
        .read_iter      = read_iter_zero,
        .write_iter     = write_iter_zero,
        .mmap           = mmap_zero,
        .get_unmapped_area = get_unmapped_area_zero,
#ifndef CONFIG_MMU
        .mmap_capabilities = zero_mmap_capabilities,
#endif
};

static const struct file_operations full_fops = {
        .llseek         = full_lseek,
        .read_iter      = read_iter_zero,
        .write          = write_full,
};

static const struct memdev {
       const char *name;
       umode_t mode;
       const struct file_operations *fops;
       fmode_t fmode;
} devlist[] = {
#ifdef CONFIG_DEVMEM
        [1] = { "mem", 0, &mem_fops, FMODE_UNSIGNED_OFFSET },
#endif
```

```
#ifdef CONFIG_DEVKMEM
         [2] = { "kmem", 0, &kmem_fops, FMODE_UNSIGNED_OFFSET },
#endif
         [3] = { "null", 0666, &null_fops, 0 },
#ifdef CONFIG_DEVPORT
         [4] = { "port", 0, &port_fops, 0 },
#endif
         [5] = { "zero", 0666, &zero_fops, 0 },
         [7] = { "full", 0666, &full_fops, 0 },
         [8] = { "random", 0666, &random_fops, 0 },
         [9] = { "urandom", 0666, &urandom_fops, 0 },
#ifdef CONFIG_PRINTK
        [11] = { "kmsg", 0644, &kmsg_fops, 0 },
#endif
};

...

static int memory_open(struct inode *inode, struct file *filp)
{
        int minor;
        const struct memdev *dev;

        minor = iminor(inode);
        if (minor >= ARRAY_SIZE(devlist))
                return -ENXIO;

        dev = &devlist[minor];
        if (!dev->fops)
                return -ENXIO;

        filp->f_op = dev->fops;
        filp->f_mode |= dev->fmode;
```

<< This (below) is the MINOR-SPECIFIC device open routine => when a device file with this major # is opened, and it has a non-null "open" member in it's FOP, the device driver itself calls this (secondary) open function.
>>

```
        if (dev->fops->open)
                return dev->fops->open(inode, filp);

        return 0;
}

…

static const struct file_operations memory_fops = {
        .open = memory_open,
        .llseek = noop_llseek,
};
```

```
...

static int __init chr_dev_init(void)
{
        int i;

        if (register_chrdev(MEM_MAJOR, "mem", &memory_fops))
                printk("unable to get major %d for memory devs\n", MEM_MAJOR);
```

<<
The kernel API register_chrdev() **updates the chrdevs table** in kernel memory (equivalent to Unix's cdevsw). The table is (simplistically) an array of structures containing the device major # and a pointer to f_op's. The f_op structure holds a single function pointer - the open method. This is the “major-wide” open performed when any device file with this major # is opened with the open(2) system call.
Note that on 2.6, with the introduction of the new device model, the device struct has changed and is more rich and complex.
>>

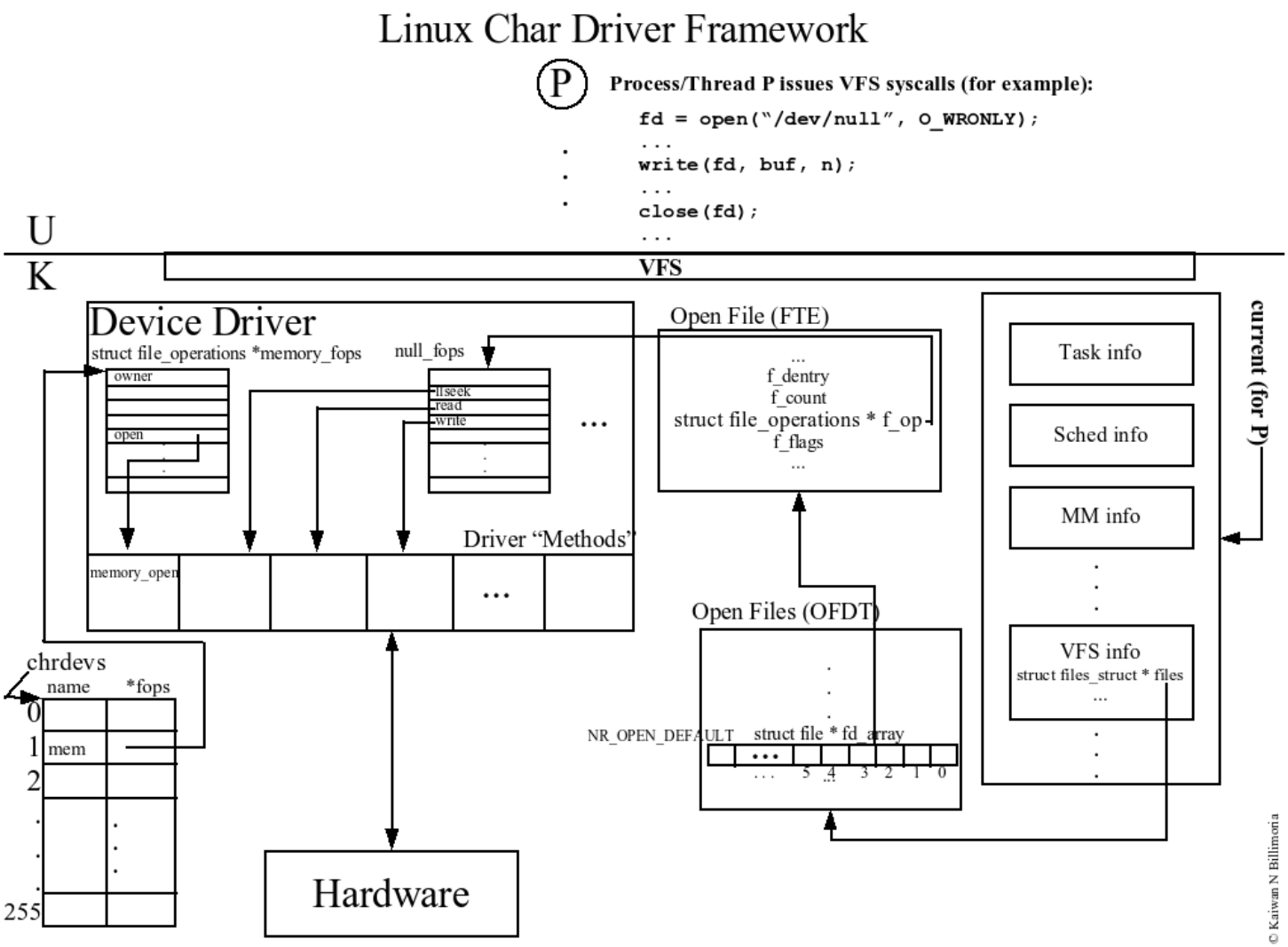
```
        << create relevant device nodes dynamically >>

        return 0;
}

fs_initcall(chr_dev_init);
```

SIDEBAR – Kernel Initialization

The fs_initcall tells the kernel to call the *chr_dev_init()* function as part of it's initialization; (fyi, the kernel init code that runs all the init calls is in *init/main.c:do_initcalls()* .

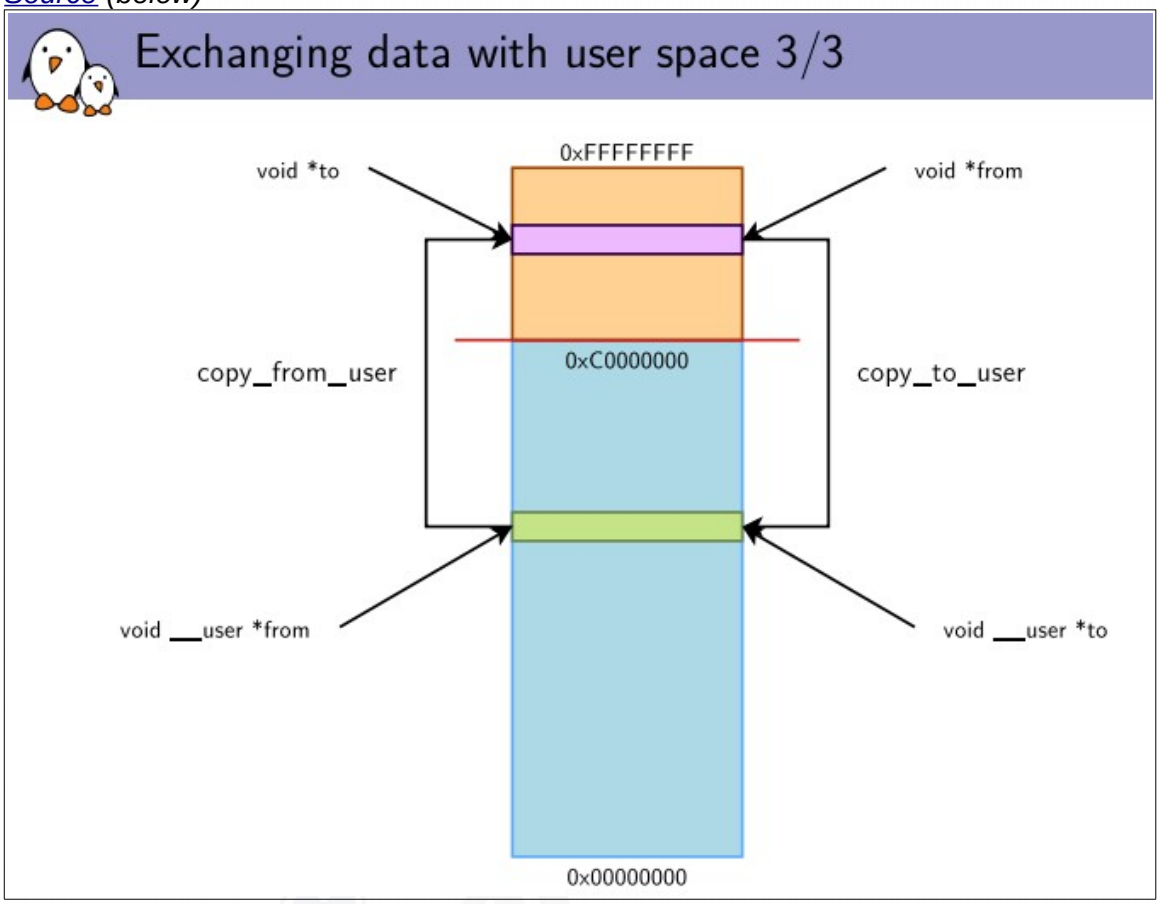
The (partial) kernel initialization call chain is:

```
start_kernel → rest_init → | →  ...
                           | → kernel_init → do_basic_setup → do_initcalls
                                    (kernel thread)
```


Linux Char Driver Framework
P
Process/Thread P issues VFS syscalls (for example):
fd = open("/dev/null", O_WRONLY);
...
write(fd, buf, n);
...
close(fd);
...
U
K
VFS
Device Driver
struct file_operations *memory_fops
null_fops
owner
open
llseek
read
write
Driver "Methods"
memory_open
Open File (FTE)
f_dentry
f_count
struct file_operations * f_op
f_flags
Open Files (OFDT)
NR_OPEN_DEFAULT
struct file * fd_array
Task info
Sched info
MM info
VFS info
struct files_struct * files
current (for P)
chrdevs
name
*fops
0
1
mem
2
255
Hardware
© Kaiwan N Billimoria

Copying User <--> Kernel Memory

Source (below)



I The Older Way with access_ok

Address verification for kernels 2.2.x and beyond is implemented by the function *access_ok*, which is declared in *asm/uaccess.h*:

```
#include <asm/uaccess.h>

#define access_ok(type,addr,size)   (__range_ok(addr,size) == 0)
```

type: should be either **VERIFY_READ** or **VERIFY_WRITE**, depending on whether the action to be performed is reading the user-space memory area or writing it. The **addr** argument holds a user-space address; **size** is a byte count.

If you need to both read and write at the given address, use VERIFY_WRITE, since it is a superset of VERIFY_READ.

Unlike most functions, access_ok returns a boolean value: **1 for success (access is OK)** and **0 for failure** (access is not OK). If it returns false, the driver will usually return **-EFAULT** to the caller.

There are a couple of interesting things to note about access_ok. First is that it does not do the complete job of verifying memory access; it only checks to see that the memory reference is in a region of memory that the process might reasonably have access to. In particular, access_ok ensures that the address **does not point** to kernel-space memory.

Second, most driver code ***need not actually call access_ok()***. The memory-access routines described next take care of that for you.

Example:

```
...
if(!access_ok(VERIFY_WRITE, buf, count)){ // 0 on failure
        ret=-EFAULT;
        goto out;
}
...
```

II Copying with copy_[from|to]_user

Drivers, and kernel code in general, cannot just transfer data to and from kernel and user-space by merely initializing/populating memory addresses (just using memcpy / memset will not work). This is as a memory fault could occour while copying data; handling the fault is non-trivial: kernel-space macros do this using an exception table and a “fixup” section. Also, the actual code is in hand-coded inline assembler to improve performance.

Linux provides two routines to correctly perform data transfer- these will typically be used within the *read()* and *write()* (and often the *ioctl()*) system call implementations of a driver.

write(): U → K : user to kernel space data transfer:

The routine is copy_from_user()

For both, include the kernel header like so:

```
//--- copy_[to|from]_user()
#include <linux/version.h>
#if LINUX_VERSION_CODE > KERNEL_VERSION(4,11,0)
```

```
#include <linux/uaccess.h>
#else
#include <asm/uaccess.h>
#endif

static inline long copy_from_user(void *to,
        const void __user * from, unsigned long n) ;
```

read(): K → U : kernel to user space data transfer:
The routine is copy_to_user().

```
static inline long copy_to_user(void __user *to,
        const void *from, unsigned long n)
```

SIDEBAR :: *"Why should we use copy_from_user?"* ***Source***

This is a famous question. Let me write my understanding of this.

The fundamental part is protection. The kernel should never let user-space access any kernel memory directly. So, there is a *access_ok()* that is called on the address to ensure that the address is a user space address and not a kernel space address, and also check if it can be read or written. It is very critical for obvious security reason.

The natural next question is why can't the kernel just complete this check and read from this address directly when needed instead of copying into its buffer?

What happens when there is a page fault when reading this address and the memory pointed is invalid. Usually, if we have a page fault inside the kernel the kernel will panic() and give up. It is definitely not a good idea to let a user-space process panic the kernel. So, if there is a page-fault during *copy_from_user* the kernel does not panic but nicely returns a error like -EFAULT to the user.

Is this not a overhead? Yes! it is. But, this is a overhead that is needed to ensure stability. This code is written in assembly. [Also: This is one of the places where the PLD instructions of the ARM are used so that this is done in the most optimal way.]
…"

Although these functions behave like normal memcpy functions, a little extra care must

be used when accessing user space from kernel code. The user pages being addressed **might not be currently present in memory**, and the page-fault handler **can put the process to sleep** while the page is being transferred into place. This happens, for example, when the page must be retrieved from swap space. The net result for the driver writer is that any function that accesses user space must be reentrant and must be able to execute concurrently with other driver functions. That's why we use semaphores to control concurrent access.

The role of the two functions is not limited to copying data to and from user-space: **they also check whether the user space pointer is valid**. If the pointer is invalid, no copy is performed; if an invalid address is encountered during the copy, on the other hand, only part of the data is copied.

In both cases, *the return value is the amount of memory **still to be copied***. The code looks for this error return, and returns **-EFAULT** to the user if it's **not 0**.

<<
Internal Implementation

```
static inline long copy_from_user(void *to,
        const void __user * from, unsigned long n)
{
    might_fault();
    if (access_ok(VERIFY_READ, from, n))
        return __copy_from_user(to, from, n);
    else
        return n;
}

static inline long copy_to_user(void __user *to,
        const void *from, unsigned long n)
{
    might_fault();
    if (access_ok(VERIFY_WRITE, to, n))
        return __copy_to_user(to, from, n);
    else
        return n;
}
```

>>

Example:

```
...
if (copy_from_user (kbuf, buf, n)) {
      ret=-EFAULT;
      goto out;
}
...
```

[OPTIONAL / FYI]

See Appendix A :: Testing the access_ok() and copy_to_user() using an application and device driver.

Source

The Linux kernel provides several functions that you can use to move system call arguments to and from user-space. Options include simple functions for basic types (such as *get_user* or *put_user*).

For moving blocks of data such as structures or arrays, you can use another set of functions: *copy_from_user* and *copy_to_user*.

Moving null-terminated strings have their own calls:
strncpy_from_user and *strlen_from_user*.

You can also test whether a user-space pointer is valid through a call to *access_ok*. These functions are defined in *linux/include/asm/uaccess.h*.

You use the *access_ok* macro to validate a user-space pointer for a given operation. This function takes the type of access (VERIFY_READ or VERIFY_WRITE), the pointer to the user-space memory block, and the size of the block (in bytes). The function returns zero on success:
int **access_ok**(type, address, size);

Moving simple types between the kernel and user-space (such as ints or longs) is accomplished easily with *get_user* and *put_user*. These macros each take a value and a pointer to a variable. The *get_user* function moves the value that the user-space address specifies (ptr) into the kernel variable specified (var).

The *put_user* function moves the value that the kernel variable (var) specifies into the user-space address (ptr). The functions return zero on success:
int **get_user**(var, ptr);
int **put_user**(var, ptr);

To move larger objects, such as structures or arrays, you can use
the *copy_from_user* and *copy_to_user* functions. These functions move an entire block of

data between user-space and the kernel. The *copy_from_user* function moves a block of data from user-space into kernel-space, and *copy_to_user* moves a block of data from the kernel into user-space:
unsigned long **copy_from_user**(void *to, const void __user *from, unsigned long n);
unsigned long **copy_to_user**(void *to, const void __user *from, unsigned long n);

Finally, you can copy a NULL-terminated string from user-space to the kernel by using the *strncpy_from_user* function. Before calling this function, you can get the size of the user-space string with a call to the *strlen_user* macro:
long **strncpy_from_user**(char *dst, const char __user *src, long count);
strlen_user(str);

These functions provide the basics for memory movement between the kernel and user-space. Some additional functions exist (such as those that reduce the amount of checking performed). You can find these functions in *uaccess.h*.

Note-
The actual copy_[to|from]_user() macros are implemented using inline assembler and are non-trivial as they have to take into account the fact that the userspace addresses might be invalid (or unmapped) and might require to be faulted in during the copy. If you'd like to dig a little deeper into how this works (x86-biased), look up “Documents/CopyUser” here:
http://kernelnewbies.org/Documents/CopyUser .

Device Drivers – An Example : Two Simple Memory Device Implementations

The 'cz' character device driver

[In this Session, the instructor will provide/perform:
-cz driver functionality background (for the 'czero' zero source and 'cnul' sink devices)
-comparison with existing /dev/zero and /dev/null devices
-code walkthrough
]

cz_enh: Enhanced zero-source functionality

The previous driver's (cz.c) zero-source implementation could only populate a user buffer of upto 1 page (typically 4Kb). An enhanced function (in cz_enh.c) that fills a user-space buffer of *any* size with NULLs is shown below (the rest of the driver remains the same and is therefore not shown again):

```
/*
 * Simple zero source implementation: just fill a buffer with zeroes
 * (for upto PAGE_SIZE bytes) & pass it back to user-space.
 *
 * This is an enhanced version of the previous driver- it populates a
 * user-space buffer of *any* size (not just to a max of 1 page as in the
 * previous driver).
 *
 */
static ssize_t czero_read(struct file *filp, char __user *buf,
                size_t count, loff_t *offp)
{
        char *zbuf;
        int mcount=count, i=0, loopcount, buf_to=0, rem=0;

        MSG( "process %s [pid %d] to read %d bytes; buf=0x%px\n",
                current->comm, current->pid, count, (unsigned int)buf );

        if (count>PAGE_SIZE)
                mcount=PAGE_SIZE

   /* kzalloc() not supported on RHEL4's kernel ver; >2.6.18 ?
    * API ref: http://gnugeneration.com/books/linux/2.6.20/kernel-api/re241.html
     */
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,6,14)
    if ( (zbuf=kzalloc (mcount, GFP_KERNEL)) == NULL) {
#else
    if ( (zbuf=kmalloc (mcount, GFP_KERNEL)) == NULL) {
```

```
#endif
        status = -ENOMEM;
        goto out_no_mem;
    }
#if LINUX_VERSION_CODE <= KERNEL_VERSION(2,6,14)
    memset (zbuf, 0, mcount);
#endif

  /* Loop abs(count/PAGE_SIZE) times */
  loopcount=count/PAGE_SIZE;
  MSG ("loopcount=%d\n",loopcount);

  for (i=0; i<loopcount; i++) {
          buf_to = (unsigned int)buf + (i*PAGE_SIZE);
          MSG ("%d: buf_to loc=0x%px zbuf=0x%px mcount=%d\n",
                  i, buf_to, (void *)zbuf, mcount);
          if( copy_to_user( (void *)buf_to, zbuf, mcount ) ) {
                  kfree( zbuf );
                  return -EFAULT;
          }
  }

  /* Remaining bytes to fill */
  rem = count - (loopcount * PAGE_SIZE);
  MSG ("rem=%d\n",rem);
  if (!rem) goto out;

  buf_to = (unsigned int)buf + (i*PAGE_SIZE);
  MSG ("%d: buf_to loc=0x%px\n",i, buf_to);
  if( copy_to_user( (void *)buf_to, zbuf, rem ) ) {
          kfree( zbuf );
          return -EFAULT;
  }

out:
        kfree( zbuf );
        return count;
}
```

Example Usage

```
# insmod cz-enh.ko
# lsmod|grep cz
cz_enh                  3876  0
# ./rd_tst
./rd_tst device_name type_of_read num_bytes
type_of_read: 0 -> read() system call
              1 -> fread() stdc lib call
# ./rd_tst cz 0 4000
Testing czero (8289) reads..
device opened: fd=3
read 4000 bytes: buf[0] is: 00
lseek failed: Illegal seek
# dmesg
...
[17185398.652000] cz:cz_open:163: Device node with minor # 1 being used
[17185398.652000] cz:czero_read:72: process rd_tst [pid 8289] to read 4000 bytes;
buf=0x0804a008
[17185398.652000] cz:czero_read:80: loopcount=0
[17185398.652000] cz:czero_read:93: rem=4000
[17185398.652000] cz:czero_read:97: 0: buf_to loc=0x0804a008
# ./rd_tst cz 0 30000
Testing czero (8292) reads..
device opened: fd=3
read 30000 bytes: buf[0] is: 00
lseek failed: Illegal seek
# dmesg
...
[17185425.004000] cz:cz_open:163: Device node with minor # 1 being used
[17185425.004000] cz:czero_read:72: process rd_tst [pid 8292] to read 30000 bytes;
buf=0x0804a008
[17185425.004000] cz:czero_read:80: loopcount=7
[17185425.004000] cz:czero_read:84: 0: buf_to loc=0x0804a008
[17185425.004000] cz:czero_read:84: 1: buf_to loc=0x0804b008
[17185425.004000] cz:czero_read:84: 2: buf_to loc=0x0804c008
[17185425.004000] cz:czero_read:84: 3: buf_to loc=0x0804d008
[17185425.004000] cz:czero_read:84: 4: buf_to loc=0x0804e008
[17185425.004000] cz:czero_read:84: 5: buf_to loc=0x0804f008
[17185425.004000] cz:czero_read:84: 6: buf_to loc=0x08050008
[17185425.004000] cz:czero_read:93: rem=1328
[17185425.004000] cz:czero_read:97: 7: buf_to loc=0x08051008
#
```

Participants can find the source code for the cz device driver in the GitHub repo.

SIDEBAR :: Using crash to see into the kernel

The superb 'crash' utility (here used in 'live dump' mode on a Fedora 29 x86-64 VM running a custom 5.0 kernel) enables us to literally peep into kernel data structures. Here, we can see the relevant file operation data structures that various devices (drivers, really), have:

```
$ sudo crash /proc/kcore <...>/vmlinux
...
WARNING: kernel relocated [832MB]: patching 105301 gdb minimal_symbol values

      KERNEL: /home/lkdc/kernels/linux-5.0-lkdc/vmlinux
    DUMPFILE: /proc/kcore
        CPUS: 2
        DATE: Tue Dec 24 12:51:25 2019
      UPTIME: 00:22:00
LOAD AVERAGE: 0.42, 0.25, 0.35
       TASKS: 170
    NODENAME: lkdc-f29
     RELEASE: 5.0.0-dbg1
     VERSION: #4 SMP Fri Jun 21 12:23:09 IST 2019
     MACHINE: x86_64  (2592 Mhz)
      MEMORY: 2 GB
         PID: 8436
     COMMAND: "crash"
        TASK: ffff8880685b8040  [THREAD_INFO: ffff8880685b8040]
         CPU: 1
       STATE: TASK_RUNNING (ACTIVE)

crash> dev -h
dev: invalid option -- 'h'
Usage:
  dev [-i | -p | -d | -D]
Enter "help dev" for details.

crash> dev
CHRDEV    NAME              CDEV        OPERATIONS
   1      mem           ffff88806b5c5a48  memory_fops
   4      /dev/vc/0     ffffffffb9178c40  console_fops
   4      tty           ffff88806a14e3c8  tty_fops
   4      ttyS          ffff888068e8e3c8  tty_fops
   5      /dev/tty      ffffffffb9177980  tty_fops
   5      /dev/console  ffffffffb91778e0  console_fops
   5      /dev/ptmx     ffffffffb9177e40  ptmx_fops
   7      vcs           ffff88806b5c50e8  vcs_fops
  10      misc          ffff88806b50c5a8  misc_fops
  13      input         ffff88806a30c918  joydev_fops
  14      sound         ffff8880347de1e8  soundcore_fops
  21      sg            ffff8880684085a8  sg_fops
  29      fb            ffff88806c2870e8  fb_fops
 116      alsa          ffff888068e52008  snd_fops
 128      ptm           ffff888068e8e968  tty_fops
```

```
 136      pts            ffff888068e8fa48  tty_fops
 162      raw            ffffffffb91831e0  raw_fops
 180      usb            ffff88806b50da48  usb_fops
 188      ttyUSB              (none)
 189      usb_device     ffffffffb9191d60  usbdev_file_operations
...
 254      gpiochip            (none)

BLKDEV    NAME                GENDISK      OPERATIONS
 259      blkext              (none)
   8      sd             ffff888068686728  sd_fops
   9      md                  (none)
  11      sr             ffff888068685dc8  sr_bdops
  65      sd                  (none)
...
crash>
```

Lookup the memory_fops structure; only the .open member is relevant!

```
memory_fops = $2 = {
  owner = 0x0,
  llseek = 0xffffffffb54a2960,
  read = 0x0,
  write = 0x0,
...
  open = 0xffffffffb5965cb0,
  flush = 0x0,
  release = 0x0,
...
crash> sym 0xffffffffb5965cb0
ffffffffb5965cb0 (t) memory_open  <...>/linux-5.0/drivers/char/mem.c: 883
crash>
```

Even points us to the file:line of code where this symbol is defined!

Sidebar

Note on register_chrdev changes in 2.6

[Extracted from the “Driver Porting” article series on lwn.net by Jonathan Corbet: http://lwn.net/Articles/49684/]

The new way

register_chrdev() continues to work as it always did, and drivers which use that function need not be changed. Unchanged drivers, however, will not be able to use the expanded device number range, or take advantage of the other features provided by the new code. Sooner or later, it is worthwhile to get to know the new interface.

The new way to register a char device range is with:

```
int register_chrdev_region(dev_t from, unsigned count,
                           char *name);
```

Here, from is the device number of the first device in the range, count is the number of device numbers to register, and name is the base name of the device (it appears in /proc/devices). The return value is zero if all goes well, and a negative error number otherwise.

Note that from is a device number, not a major number. This interface allows the registration of an arbitrary range of device numbers, starting from anywhere. So the from argument specifies both the beginning major and minor number. If the count argument exceeds the number of minor numbers available, the allocation will continue on into the next major number; this is a design feature.

register_chrdev_region() works if you know which major device number you wish to use. If, instead, your driver expects to work with dynamic major number allocation, it should use:

```
int alloc_chrdev_region(dev_t *dev, unsigned baseminor,
                        unsigned count, char *name);
```

In this case, dev is an output-only parameter which will be set to the first device number of the allocated range. The input parameters are baseminor, the first minor number to use (usually zero); count, the number of device numbers to allocate; and name, the base name of the device. Once again, the return value is zero or a negative error code.

For your convenience (and a quick start), a “character device driver template” kernel module has been provided on the GitHub repo for this course. Try it out!

```
$ cat Makefile
# Makefile for 2.6 kernel cz driver
# This Makefile idiom from the Rubini & Corbet (LDD3) book.

ifneq ($(KERNELRELEASE),)
obj-m       := cz.o
EXTRA_CFLAGS += -DDEBUG

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

default:
        $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
install:
       $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules_install
clean:
       $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) clean
endif
$
```

Notice in the Makefile above:

1. Adding
EXTRA_CFLAGS += -DDEBUG
defines the symbol 'DEBUG' in the module.

2. The 'install' target in the Makefile lets one add the kernel module to the /lib/modules/`uname -r`/extra folder.
You need to run this as root user.

Eg.
In the Makefile include the 'install' target:

```
       ...
       install:
              $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules_install
       …

# make install
make -C /lib/modules/2.6.24-23-generic/build SUBDIRS=/mnt/<...>/cz modules_install
make[1]: Entering directory `/usr/src/linux-headers-2.6.24-23-generic'
  INSTALL /mnt/<...>/cz.ko
  INSTALL /mnt/<...>/cz_enh.ko
  DEPMOD  2.6.24-23-generic
make[1]: Leaving directory `/usr/src/linux-headers-2.6.24-23-generic'
```

```
# ls -l /lib/modules/2.6.24-23-generic/extra/
total 220
-rw-r--r-- 1 root root 107067 2009-01-22 15:43 cz_enh.ko
-rw-r--r-- 1 root root 106379 2009-01-22 15:43 cz.ko
#
# modprobe cz_enh
FATAL: Module cz_enh not found.
# depmod -a                ← generates the 'modules.dep' file
# modprobe cz_enh
# lsmod |head
Module                  Size  Used by
cz_enh                  5892  0
e1000                 126016  0
...
#
```

Q. What exactly is the *modules.dep* file for?
A. See "OS TUTORIALS: LINUX KERNEL MODULE DEPENDENCY ENTRIES (Modules.dep)"

Useful Notes-

1. Any kernel modules you'd like loaded at boot time? Just add their names to */etc/modules* (Debian-specific?).

2. Source:
--snip--

```
=== 5. Module Installation

Modules which are included in the kernel are installed in the
directory:

        /lib/modules/$(KERNELRELEASE)/kernel/

And external modules are installed in:

        /lib/modules/$(KERNELRELEASE)/extra/

--- 5.1 INSTALL_MOD_PATH

        Above are the default directories but as always some level of
        customization is possible. A prefix can be added to the
        installation path using the variable INSTALL_MOD_PATH:

                $ make INSTALL_MOD_PATH=/frodo modules_install
```

```
                => Install dir:
                    /frodo/lib/modules/$(KERNELRELEASE)/kernel/

        INSTALL_MOD_PATH may be set as an ordinary shell variable or,
        as shown above, can be specified on the command line when
        calling "make." This has effect when installing both in-tree
        and out-of-tree modules.

--- 5.2 INSTALL_MOD_DIR

        External modules are by default installed to a directory under
        /lib/modules/$(KERNELRELEASE)/extra/, but you may wish to
        locate modules for a specific functionality in a separate
        directory. For this purpose, use INSTALL_MOD_DIR to specify an
        alternative name to "extra."

                $ make INSTALL_MOD_DIR=gandalf -C $KDIR \
                       M=$PWD modules_install
                => Install dir: /lib/modules/$(KERNELRELEASE)/gandalf/
```

--snip--

Device Nodes Management

It's important to realize that the device driver itself does not create the device nodes (*/dev/...* files) – they have to be created.

The mknod(1) utility achieves this (you have to run it as root). The disadvantage of using dynamic major assignment, however, is now obvious: you cannot create the device nodes in advance with mknod because you don't know which major number will be assigned. The solution to this is straight-foward: /proc/devices will hold your driver's major number – just grep for it by name and filter out the rest, getting the major number.

Eg.:

```
# insmod cz.ko
# grep cz /proc/devices
254 cz
#
```

The **shell script** (optional) shown below, does just this: you can use it to create and refresh stale device nodes in a dynamic fashion.

$ cat load_cz.sh

```
#!/bin/bash
# load_cz.sh
# Helper script to load the "cz" or "cz-enh" kernel modules,
# also creating the device nodes as necessary.
#
# Usage:
# load_cz.sh [-option] module ; where option is one of:
#       -c :    create and load the device files DEV1 and DEV2
#       -l :    only load the device module into the kernel
#       --help: show this help screen
#

# Configuration variables
DEV1="/dev/czero"
DEV2="/dev/cnul"

PERMS="666"     # for the device files DEV1 and DEV2
MINOR_CZ=1
MINOR_CNUL=2

# Refresh the device nodes
function creat_dev() {
        echo "Creating device files $DEV1 and $DEV2.."

        # remove any stale instances
        rm $DEV1 2>/dev/null
        rm $DEV2 2>/dev/null

        # Get dynamic major
        # Assumption: the name of the driver has "cz"
        MAJOR=$(grep -w "cz" /proc/devices | awk '{print $1}')
        echo "$0: MAJOR number is $MAJOR"

        mknod -m$PERMS $DEV1 c $MAJOR $MINOR_CZ
        if [ $? -ne 0 ]; then
                echo "$DEV1 device creation mknod failure.."
        fi

        /bin/mknod -m$PERMS $DEV2 c $MAJOR $MINOR_CNUL || {
                echo "$DEV2 device creation mknod failure.."
        }
}

function load_dev() {
        /sbin/rmmod $KMOD 2>/dev/null   # in case it's already loaded

        /sbin/insmod $KMOD || {
                echo "insmod failure.."
                exit 1
        }

        echo "Module $KMOD successfully inserted"
```

```
        echo "Looking up last 5 printk's with dmesg.."
        /bin/dmesg | tail -n5
}

function usage() {
        echo "Usage help:: \
        $0 [-option] module ;
 where option is one of:
        -c :    create and load the device files DEV1 and DEV2
        -l :    only load the device module [module] into the kernel
        --help: show this help screen
 and module is the filename of the module."
        exit 1
}

# main here

if [ "$(id -u)" != "0" ]; then
        echo "Must be root to load/unload kernel modules"
        exit 1
fi

if [ $# -ne 2 ]; then
        usage
fi

KMOD=$2
if [ ! -e $KMOD ]; then
        echo "$0: $KMOD not a valid file/pathname."
        exit 1
fi

case "$1" in
   --help)
        usage
        ;;
   -c)
        echo "Loading module and creating device files for device $DEV now.."
        load_dev
        creat_dev
        ;;      # falls thru to the load..
   -l)
        echo "Loading device $DEV now.."
        load_dev
        exit 0
        ;;
   *)
        usage
        ;;
esac

exit 0
$
```

* FYI, there *is* a way to directly invoke user-space programs (binaries) from within the kernel (LKM or inline): the kernel provides the **`call_usermodehelper()`** API for this purpose.

Abusing it is definitely not recommended.

kernel/umh.c

```
/**
 * call_usermodehelper() - prepare and start a usermode application
 * @path: path to usermode executable
 * @argv: arg vector for process
 * @envp: environment for process
 * @wait: wait for the application to finish and return status.
 *        when UMH_NO_WAIT don't wait at all, but you get no useful
 *        error back when the program couldn't be exec'ed. This makes it
 *        safe to call from interrupt context.
 *
 * This function is the equivalent to use call_usermodehelper_setup()
 * and call_usermodehelper_exec().
 */
int call_usermodehelper(const char *path, char **argv, char **envp, int
wait) ...
```

A good tutorial on the *`call_usermodehelper()`* API usage can be found here.

A Brief Note on udev

* On 2.6 Linux onward, the */dev* directory is now a "pseudo" folder (a RAMdisk), managed by a facility called **udev**. udev allows Linux users to have a *dynamic /dev* directory and it provides the ability to have persistent device names.

Early Linux distributions used a static 'dev' directory node and static device files. But it grew too big to manage; RedHat 9.0 had 18,000 device files under */dev* !

udev on Wikipedia

Hotplugging with udev, M Opdenacker
Related: Note on Modalias

Some Nifty udev Rules and Examples

Writing udev rules
SO - How are kernel uevents propogated to user-space

A quick example:

```
$ lspci |grep -i ethernet
00:19.0 Ethernet controller: Intel Corporation 82579LM Gigabit Network
Connection (rev 04)
$
```

00:19.0 => PCI bus 0, device # 19.0
Verify with '-n' option:

```
$ lspci -n |grep "19\.0"
00:19.0 0200: 8086:1502 (rev 04)
$
```

VID (vendor id) 0x8086 (Intel), device id 1502.

```
$ ls /sys/bus/pci/devices/
0000:00:00.0@  0000:00:19.0@  0000:00:1c.1@  0000:00:1d.0@
0000:03:00.0@
0000:00:02.0@  0000:00:1a.0@  0000:00:1c.3@  0000:00:1f.0@
0000:0d:00.0@
0000:00:16.0@  0000:00:1b.0@  0000:00:1c.4@  0000:00:1f.2@
0000:0e:00.0@
0000:00:16.3@  0000:00:1c.0@  0000:00:1c.6@  0000:00:1f.3@
$ ls /sys/bus/pci/devices/0000\:00\:19.0/
broken_parity_status      driver@          modalias   remove
subsystem@
class                     driver_override  msi_bus    rescan
subsystem_device
config                    enable           msi_irqs/  reset
subsystem_vendor
consistent_dma_mask_bits  firmware_node@   net/       resource   uevent
d3cold_allowed            irq              numa_node  resource0  vendor
device                    local_cpulist    power/     resource1
dma_mask_bits             local_cpus       ptp/       resource2
$ cat /sys/bus/pci/devices/0000\:00\:19.0/modalias
pci:v00008086d00001502sv000017AAsd000021CEbc02sc00i00
$
```

Interpreting the modalias:

pci obviously implies a PCI device.
The string following it
v00008086d00001502sv000017AAsd000021CEbc02sc00i00

is just a representation of a structured format from hardware.

Source

...
Well, hey, I can see pci! I recognize that, but what's all this gibberish at the end? This gibberish is actually structured data. You will notice a repeating letter/number scheme. Let's split this apart to make it easier to read:

```
v  00008086       <<  v: vendorID >>
d  000024DB       <<  d: deviceID >>
sv 0000103C       << sv: subsystem vendorID >>
sd 0000006A       << sd: subsystem deviceID >>
bc 01             << bc: base class >>
sc 01             << sc: sub class >>
i  8A             <<  i: programming interface >>
```

Each of these identifiers, and corresponding hex numbers represent some of the info that a given device exposes. For starters, **v** is the *vendor id* and **d** is the *device id* - these are very standard numbers, and are the same exact numbers that tools like **hwd** uses to lookup a device. You can even find websites to look up specific hardware identification based on the vendor and device ids, for instance, **http://www.pcidatabase.com/**

...

For the record, sv and sd are "subsystem" versions of both vendor and device. A majority of the time these are ignored. They are mainly used by the hardware developers to distinguish slight differences in the inner workings which do not change the device as a whole.

bc (base class) and sc (sub class) are used to create the "Class" listed by lspci, in order "bcsc". This is the device class, which is fairly generic. In this case, the "class" is looked up in the normal lspci output. We can see that "Class 0101" maps to "IDE Interface" (lspci also looks up the vendor and device id's - 8086 maps to "Intel Corp." and 24DB maps to 'Unknown Device', hehe)

i is the "Programming interface", which is only meaningful for a few devices classes.

So now we can interpret the modlalias example!

```
pci:v00008086d00001502sv000017AAsd000021CEbc02sc00i00
```

PCI device.
v00008086 : Vendor ID
d00001502 : Device ID
sv000017AA : Subsystem vendor ID
sd000021CE : Subsystem deviceD
bc02 : Base Class
sc00 : Sub Class
i00 : programming Interface

Key point:
depmod builds the "map" files which tells modprobe which kernel module to insert. The key one is **modalias**. It contains "aliases" or "other-names" for modules. For eg. for the Intel ethernet controller above:

```
$ grep -i e1000 /lib/modules/3.19.0-65-generic/modules.alias
alias pci:v00008086d00002E6Esv*sd*bc*sc*i* e1000
alias pci:v00008086d000010B5sv*sd*bc*sc*i* e1000
alias pci:v00008086d00001099sv*sd*bc*sc*i* e1000
alias pci:v00008086d0000108Asv*sd*bc*sc*i* e1000
alias pci:v00008086d0000107Csv*sd*bc*sc*i* e1000
alias pci:v00008086d0000107Bsv*sd*bc*sc*i* e1000
alias pci:v00008086d0000107Asv*sd*bc*sc*i* e1000
alias pci:v00008086d00001079sv*sd*bc*sc*i* e1000
alias pci:v00008086d00001078sv*sd*bc*sc*i* e1000
alias pci:v00008086d00001077sv*sd*bc*sc*i* e1000
alias pci:v00008086d00001076sv*sd*bc*sc*i* e1000
alias pci:v00008086d00001075sv*sd*bc*sc*i* e1000
…
…
$ grep -i e1000 /lib/modules/3.19.0-65-generic/modules.alias |wc -l
101
$
```

There are 101 aliases!

Notice that each one of them has the alias "e1000". This of course implies that when the kernel detects any device matching any of the modalias strings above (note the wildcards too), it sends udevd an event: the event will contain information encoded into an environment variable "MODALIAS"; this will hold the exact hardware modalias. Udev will then consult it's rules and thus match it to a driver – in this example, the Intel e1000 driver, and invoke "modprobe e1000"!

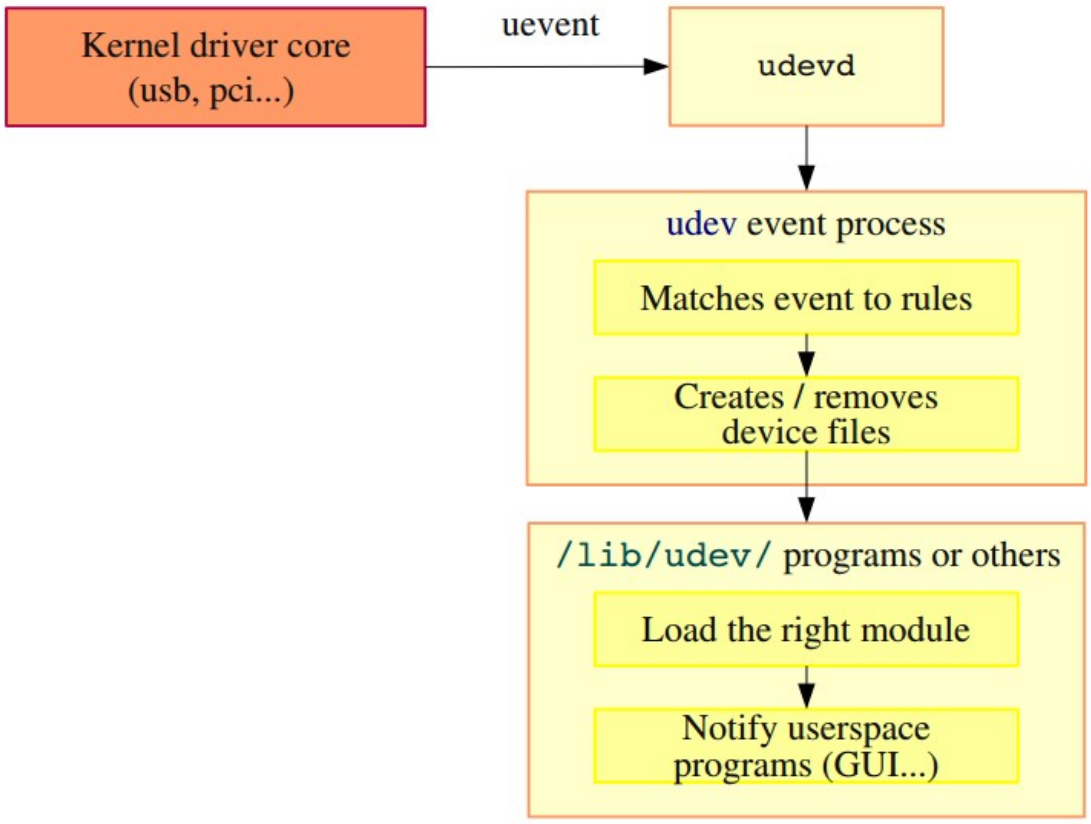
Another eg.: *Source*

- MODALIAS environment variable example (USB mouse):
  `MODALIAS=usb:v046DpC03Ed2000dc00dsc00dp00ic03isc01ip02`
- Matching line in `/lib/modules/<version>/modules.alias`:
  `alias usb:v*p*d*dc*dsc*dp*ic03isc01ip02* usbmouse`

Thus it loads the “usbmouse” kernel module.

How do we know which kernel modules drive which devices?

That's simple: the kernel driver itself provides that information (exporting it to the bus interface (like PCI/USB)). Depmod publishes this information into the *module.alias* file(s).



udev Summary – Typical Operation *Source*

mdev: Busybox lightweight udev implementation.

[OPTIONAL / FYI]

[The following section is from the book "Linux Device Drivers", 3rd Edition by A Rubini, J Corbet and GK-Hartman (O'Reilly & Associates).]

File Operations

The following list introduces **all the operations that an application can invoke on a device**. We've tried to keep the list brief so it can be used as a reference, merely **summarizing each operation and the default kernel behavior when a NULL pointer** is used. You can skip over this list on your first reading and return to it later.

As you read through the list of file_operations methods, you will note that a number of parameters include the string `__user` . This annotation is a form of *documentation*, noting that a pointer is a user-space address that cannot be directly dereferenced. For normal compilation, `__user` has no effect, but it can be used by external checking software to find misuse of user-space addresses.

```
struct module *owner
```
The first file_operations field is not an operation at all; it is a pointer to the module that "owns" the structure. This field is used to prevent the module from being unloaded while its operations are in use. Almost all the time, it is simply initialized to `THIS_MODULE`, a macro defined in <linux/module.h>.

```
loff_t (*llseek) (struct file *, loff_t, int);
```
The llseek method is used to change the current read/write position in a file, and the new position is returned as a (positive) return value. The loff_t parameter is a "long offset" and is at least 64 bits wide even on 32-bit platforms. Errors are signaled by a negative return value. If this function pointer is NULL, seek calls will modify the position counter in the file structure (described in the section "The file Structure") in potentially unpredictable ways.
Set to "`no_llseek`" if not required.

```
ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
```
Used to retrieve data from the device. A null pointer in this position causes the read system call to fail with -EINVAL ("Invalid argument"). A nonnegative return value represents the number of bytes successfully read (the return value is a "signed size" type, usually the native integer type for the target platform).

```
ssize_t (*aio_read)(struct kiocb *, char __user *, size_t,
  loff_t);
```
Initiates an asynchronous read—a read operation that might not complete before the

function returns. If this method is NULL, all operations will be processed (synchronously) by read instead.

```
ssize_t (*write) (struct file *, const char __user *, size_t,
  loff_t *);
```

Sends data to the device. If NULL, -EINVAL is returned to the program calling the write system call. The return value, if nonnegative, represents the number of bytes successfully written.

```
ssize_t (*aio_write)(struct kiocb *, const char __user *,
size_t,
  loff_t *);
```

Initiates an asynchronous write operation on the device.

```
int (*readdir) (struct file *, void *, filldir_t);
```

This field should be NULL for device files; it is used for reading directories and is useful only for filesystems.

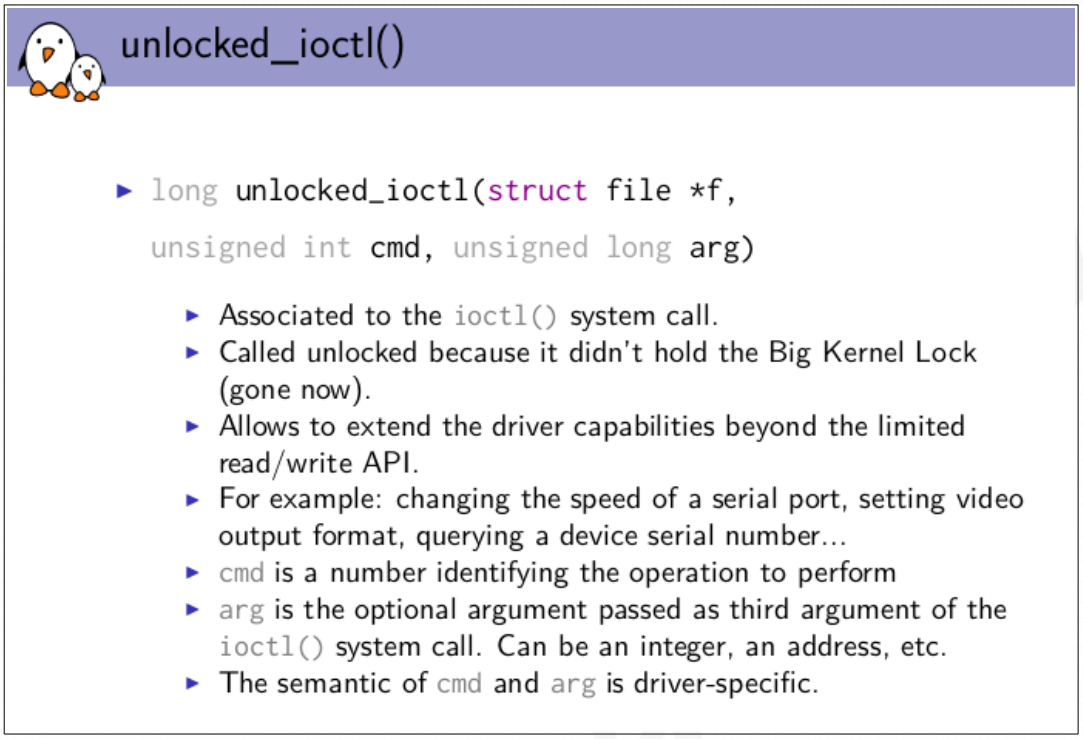
```
unsigned int (*poll) (struct file *, struct poll_table_struct *);
```

The poll method is the back end of three system calls: poll, epoll, and select, all of which are used to query whether a read or write to one or more file descriptors would block. The poll method should return a bit mask indicating whether nonblocking reads or writes are possible, and, possibly, provide the kernel with information that can be used to put the calling process to sleep until I/O becomes possible. If a driver leaves its poll method NULL, the device is assumed to be both readable and writable without blocking.

The **ioctl** system call offers a way to issue device-specific commands (such as formatting a track of a floppy disk, which is neither reading nor writing). Additionally, a few ioctl commands are recognized by the kernel without referring to the fops table. If the device doesn't provide an ioctl method, the system call returns an error for any request that isn't predefined (-ENOTTY, "No such ioctl for device").

```
<< int (*ioctl) (struct inode *, struct file *, unsigned int,
  unsigned long); : OLDER >>
long (*unlocked_ioctl) (struct file *, unsigned int, unsigned
          long);
long (*compat_ioctl) (struct file *, unsigned int, unsigned
          long);

<< BKL (Big Kernel Lock) finally removed in 2.6.36 - hence, the
unlocked_ioctl()! Use it, like so:
```

```
#include <linux/version.h>
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,36)
...
```




```
>>
```

```
int (*mmap) (struct file *, struct vm_area_struct *);
```
mmap is used to request a mapping of device memory to a process's address space. If this method is NULL, the mmap system call returns -ENODEV.

```
int (*open) (struct inode *, struct file *);
```
Though this is always the first operation performed on the device file, the driver is not required to declare a corresponding method. If this entry is NULL, opening the device always succeeds, but your driver isn't notified.

```
int (*flush) (struct file *);
```
The flush operation is invoked when a process closes its copy of a file descriptor for a device; it should execute (and wait for) any outstanding operations on the device. This must not be confused with the fsync operation requested by user programs. Currently, flush is used in very few drivers; the SCSI tape driver uses it, for example, to ensure that all data written makes it to the tape before the device is closed. If flush is NULL, the

kernel simply ignores the user application request.

```
int (*release) (struct inode *, struct file *);
```
This operation is invoked when the file structure is being released. Like open, release can be NULL.*

** Note that release isn't invoked every time a process calls close. Whenever a file structure is shared (for example, after a fork or a dup), release won't be invoked until all copies are closed. If you need to flush pending data when any copy is closed, you should implement the flush method.*

```
int (*fsync) (struct file *, struct dentry *, int);
```
This method is the back end of the fsync system call, which a user calls to flush any pending data. If this pointer is NULL, the system call returns -EINVAL.

```
int (*aio_fsync)(struct kiocb *, int);
```
This is the asynchronous version of the fsync method.

```
int (*fasync) (int, struct file *, int);
```
This operation is used to notify the device of a change in its FASYNC flag. Asynchronous
notification is an advanced topic and is described in Chapter 6. The field can be NULL if the driver doesn't support asynchronous notification.

```
int (*lock) (struct file *, int, struct file_lock *);
```
The lock method is used to implement file locking; locking is an indispensable feature for regular files but is almost never implemented by device drivers.

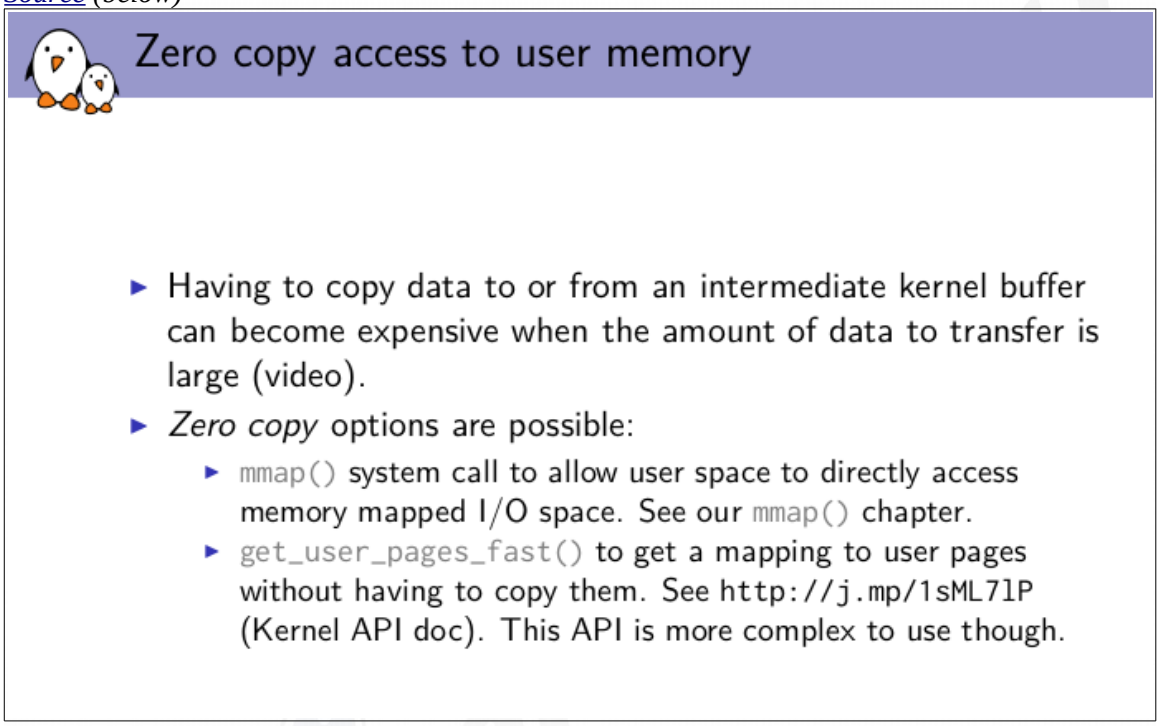
```
ssize_t (*readv) (struct file *, const struct iovec *, unsigned
  long, loff_t *);
ssize_t (*writev) (struct file *, const struct iovec *, unsigned
  long, loff_t *);
```
These methods implement scatter/gather read and write operations. Applications occasionally need to do a single read or write operation involving multiple memory areas; these system calls allow them to do so without forcing extra copy operations on the data. If these function pointers are left NULL, the read and write methods are called (perhaps more than once) instead.

```
ssize_t (*sendfile)(struct file *, loff_t *, size_t, read_actor_t,
void *);
```
This method implements the read side of the sendfile system call, which moves the data from one file descriptor to another with a minimum of copying. It is used, for example, by a web server that needs to send the contents of a file out a network connection. Device drivers usually leave sendfile NULL.

<< Useful:
A powerful feature of the modern Linux kernel and many drivers that support sophisticated DMA operations is the so-called "zero-copy" implementation.

Essentially, it gives applications that perform large data transfers between user – kernel - space a large speedup.

Source (below)



Resources:

Efficient data transfer through zero copy
www.ibm.com/developerworks/linux/library/j-zerocopy/

Zero Copy I: User-Mode Perspective
www.linuxjournal.com/article/6345

>>

```
ssize_t (*sendpage) (struct file *, struct page *, int, size_t,
loff_t *, int);
```

sendpage is the other half of sendfile; it is called by the kernel to send data, one page at a time, to the corresponding file. Device drivers do not usually implement sendpage.

```
unsigned long (*get_unmapped_area)(struct file *, unsigned long,
  unsigned long, unsigned long, unsigned long);
```

The purpose of this method is to find a suitable location in the process's address space to map in a memory segment on the underlying device. This task is normally performed by the memory management code; this method exists to allow drivers to enforce any alignment requirements a particular device may have. Most drivers can leave this method NULL.

```
int (*check_flags)(int)
```

This method allows a module to check the flags passed to an fcntl(F_SETFL...) call.

```
int (*dir_notify)(struct file *, unsigned long);
```

This method is invoked when an application uses fcntl to request directory change notifications. It is useful only to filesystems; drivers need not implement dir_notify.

User-Kernel Interfaces

<<

Warning!!

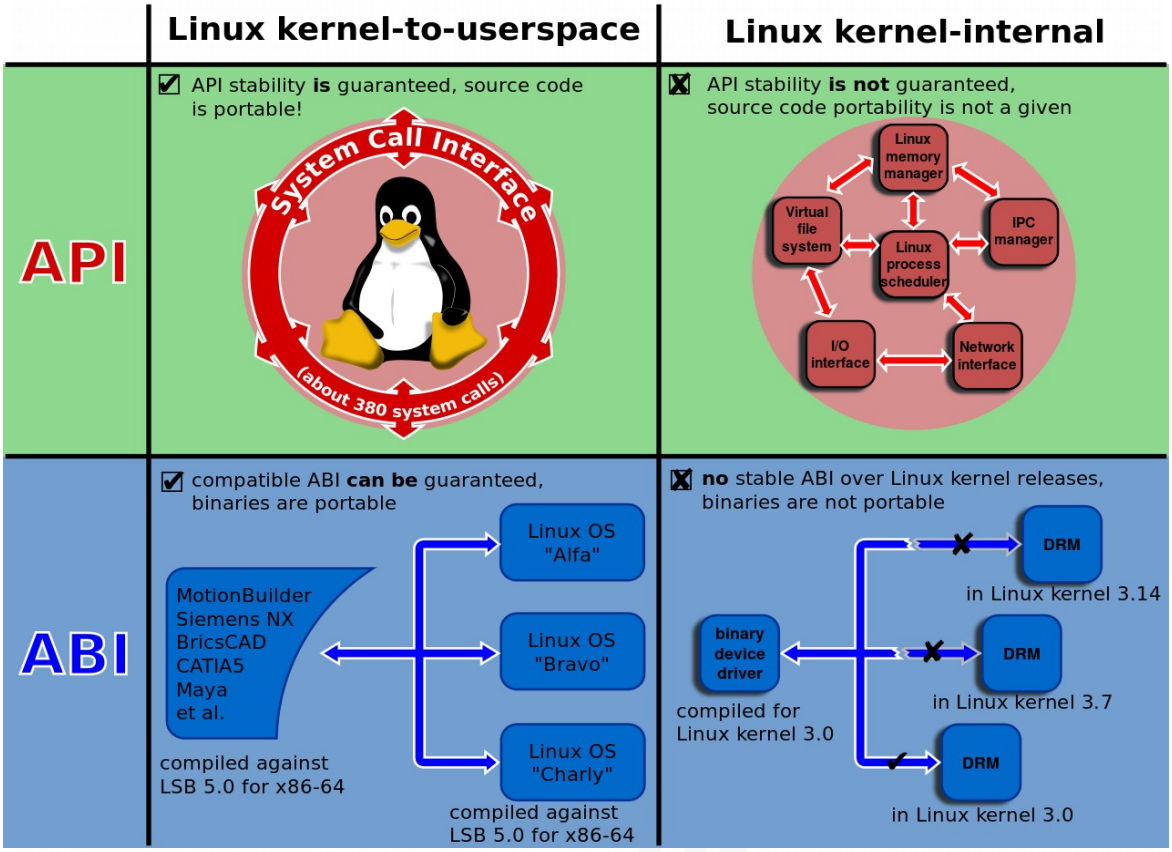
Procfs for anything other than in-kernel use is considered wrong usage and deprecated! We Strongly suggest you use other interfaces (debugfs, sysfs or netlink sockets) in place of procfs for driver-based / custom interfaces.

>>

<<

A related topic: what is an ABI? (Application Binary Interface).

API and ABI in Linux: *Source*



In-kernel API[edit]

There are a couple of kernel internal APIs utilized between the different subsystems and subsystems of subsystems. Some of them have been kept stable over several releases, other have not. There are no guarantees regarding the in-kernel APIs. Maintainers and contributors are free to augment or change them at any time.

…

In-kernel ABI[edit]

Strong interests in defining and maintaining a stable in-kernel ABI over several releases have been voiced repeatedly over time by parties, e.g. hardware manufactures, writing proprietary kernel modules and distributing binary-only software, e.g. device drivers.

By explicit choice the Linux kernel does not maintain a stable in-kernel ABI. The reasons are purely technical and not philosophical. Due to the absence of an in-kernel ABI that has to be kept stable over releases, the Linux kernel can keep a much higher rate of development.

Procfs is NOT an ABI !

>>

User-Kernel Interfaces

Source: Understanding Linux Network Internals, Benvenuti, O'Reilly

The kernel exports internal information to user space via different interfaces. Besides the classic set of system calls the application programmer can use to ask for specific information, there are three special interfaces, two of which are virtual filesystems:

procfs (/proc filesystem)

This is a virtual filesystem, usually mounted in /proc, that allows the kernel to export internal information to user space in the form of files. The files don't actually exist on disk, but they can be read through cat or more and written to with the > shell redirector; they even can be assigned permission like real files.

The components of the kernel that create these files can therefore say who can read from or write to any file. Directories cannot be written (i.e., no user can add or remove a file or a directory to or from any directory in /proc).

The default kernel that comes with most (if not all) Linux distributions includes support for procfs. It cannot be compiled as a module.

The associated kernel option from the configuration menu is
"Filesystems → Pseudo filesystems → /proc file system support."

sysctl (/proc/sys directory)

This interface allows user space to read and modify the value of kernel variables. You cannot use it for every kernel variable: the kernel has to explicitly say what variables are visible through this interface. From user space, you can access the variables exported by sysctl in two ways.

One is the sysctl system call (see man sysctl) and the other one is procfs. When the kernel has support for procfs, it adds a special directory (/proc/sys) to /proc that includes a file for each kernel variable exported by sysctl.

The sysctl command that comes with the procps package can be used to config-ure variables exported with the sysctl interface. The command talks to the kernel by writing to /proc/sys.

The default kernel that comes with most (if not all) Linux distributions includes support for sysctl. It cannot be compiled as a module. The associated kernel option from the configuration menu is *"General setup → Sysctl support."*

sysfs (/sys filesystem)

procfs and sysctl have been abused over the years, and this has led to the introduction of a newer filesystem: sysfs. sysfs exports plenty of information in a very clean and organized way. You can expect part of the information currently exported with sysctl to migrate to sysfs.

sysfs is available only with kernels starting at 2.6. The default kernel that comes with most (if not all) Linux distributions includes support for sysfs. It cannot be compiled as a module.

The associated kernel option from the configuration menu is
"Filesystems → Pseudo filesystems → sysfs filesystem support (NEW)."
The option is visible only if you first enable the following option:
"General setup
→ Configure standard kernel features (for small systems)."

<< An example character driver that employs sysfs (to create nodes) is described below – the "parrot" driver. >>

You also use the following interfaces to send commands to the kernel, either to configure something or to dump the configuration of something else:

ioctl system call

The ioctl (input/output control) system call operates on a file and is usually used to implement operations needed by special devices that are not provided by the standard filesystem calls. ioctl can be passed a socket descriptor too, as returned by the socket system call, and that is how it is used by the networking code. This interface is used by old-generation commands like ifconfig and route, among others.

Netlink socket

This is the newest and preferred mechanism for networking applications to communicate with the kernel. Most commands in the IPROUTE2 package use it.

Netlink represents for Linux what the routing socket represents in the BSD world. Most network kernel features can be configured using either Netlink or ioctl interfaces, because the kernel supports both the newer configuration tools (IPROUTE2) and the legacy ones (ifconfig, route, etc.).

procfs Versus sysctl

Both procfs and sysctl export kernel-internal information, but procfs mainly exports read-only data, while most sysctl information is writable too (but only by the superuser).

As far as exporting read-only data, the choice between procfs and sysctl depends on how much information is supposed to be exported. Files associated with a simple kernel variable or data structure are exported with sysctl. The others, which are associated with more complex data structures and may need special formatting, are exported with procfs. Examples of the latter category are caches and statistics.

…

Netlink

The Netlink socket, well described in RFC 3549, represents the preferred interface between user space and kernel for IP networking configuration. Netlink can also be used as an intrakernel messaging system as well as between multiple user-space processes.

With Netlink sockets you can use the standard socket APIs to open, close, transmit on, and receive from a socket. Let's quickly review the prototype of the socket system call:

int socket(int domain, int type, int protocol)

For details on what the three arguments are initialized to with TCP/IP sockets (i.e., domain PF_INET), you can use the *man socket* command.

As with any other socket, when you open a Netlink socket, you need to provide the domain, type, and protocol arguments. Netlink uses the new PF_NETLINK protocol family (domain), supports only the SOCK_DGRAM type, and defines several protocols, each one used for a different component (or a set of components) of the networking stack.

For example, the NETLINK_ROUTE protocol is used for most networking features, such as routing and neighboring protocols, and NETLINK_FIREWALL is used for the firewall (Netfilter). The Netlink protocols are listed in the NETLINK_XXX enumeration list in *include/linux/netlink.h*.

With Netlink sockets, endpoints are usually identified by the ID of the process that

opened the sockets (PID), where the special value 0 identifies the kernel. Among Netlink's features is the ability to send both unicast and multicast messages: the destination endpoint address can be a PID, a multicast group ID, or a combination of the two. The kernel defines Netlink multicast groups for the purpose of sending out notifications about particular kinds of events, and user programs can register to those groups if they are interested in them. The groups are listed in the enumeration list RTMGRP_XXX in include/linux/rtnetlink.h. Among them are the RTMGRP_IPV4_ROUTE and RTMGRP_NEIGH groups, used respectively for notifications regarding changes to the
routing tables and to the L3-to-L2 address mappings. …

Another interesting feature is the ability to send both positive and negative acknowledgments.

One of the advantages of Netlink over other user-kernel interfaces such as ioctl is that the kernel can initiate a transmission instead of just returning information in answer to user-space requests.

Serializing Configuration Changes

Any time you apply a configuration change, the handler that takes care of it inside the kernel acquires a semaphore (rtnl_sem) that ensures exclusive access to the data structures that store the networking configuration. This is true regardless of whether the configuration is applied via ioctl or Netlink.

A Quick Note on Sysfs, by Greg-Kroah Hartman – Source.

"sysfs is a new virtual filesystem in the 2.5 kernel that shows all of the driver, bus and class information for devices presently in the system. It also shows a lot of other information not directly related to drivers, including all of the filesystem types, all block devices, the CPU information and even a list of firmware primitives presently in the system. Over time, most of the information in the /proc filesystem that does not have to do with processes, which is what /proc started out for, has been or will be converted and moved into sysfs.

The advantages of sysfs over procfs are:

- A cleaner, well-documented programming interface.
- Directories and files are cleaned up automatically for the developer when a device is removed from the system.

- A one-item-per-file rule is enforced, which makes for a cleaner user interface.
- Internally, it plays more nicely with the kernel and does not re-implement a lot of filesystem code, sharing much logic with the libfs code.

sysfs should be mounted at /sys in the system, which can be done from the command line by running the following statement as root:
mount -t sysfs none /sys

To have it always be mounted by the system at boot time, the following line should be added to the /etc/fstab file:
none /sys sysfs defaults 0 0

I recommend poking around the deep directory tree in sysfs to try to get a better understanding of the different devices in the system and their relationship to each other."

For more details on the device model and the creation of sysfs entries, etc please see Chapter 14 "The Linux Device Model", LDD3*.
* LDD3 : "LINUX Device Drivers" by Rubini, Corbet and Kroah-Hartman, 3rd Ed, O'Reilly and Associates.
Also see:
Linux 2.6 Device Model

<<
Obviously, the "data" content of the pseudo-files under /proc are generated on the fly; the code that does this is under *fs/proc*. For example, to see (machine-format) statistics of a process:

```
$ cat /proc/1080/stat
1080 (emulator) S 858 1080 858 34818 10862 4202496 39942 0 1575 0 194690
68610 0 0 20 0 2 0 9249854 162009088 10609 4294967295 134512640
136158672 3221112384 3221095868 3086189584 0 0 4096 24578 4294967295 0 0
17 0 0 0 0 0 0
$
```

The code that generates this information can be seen here: *fs/proc/array.c:do_task_stat()* (or look it up on LXR here).
>>

Creating our own sysfs Entry Points ***[OPTIONAL]***

<< Participants will find the demo driver mentioned below – the "parrot driver" as part of the source provided. >>

---

http://pete.akeo.ie/2011/08/writing-linux-device-driver-for-kernels.html

---

debugfs

A useful code-based debug facility that is similar to procfs is the so-called "debugfs" filesystem. So, finally, debugfs is considered the "correct" place to put in all that debug-related code.
Using it is simple. This tutorial quickly covers the meat of it:
https://bitbucket.org/chadversary/debugfs-tutorial

(Participant's should download, extract, read and follow the Readme.txt 's ; this will give you useful hands-on experience using 'debugfs' from the kernel developer / debugger viewpoint.)

Very useful, please read:
Documentation/filesystems/debugfs.txt

Netlink Sockets

Source: "Kernel Korner - Why and How to Use Netlink Socket"

"Netlink socket is a special IPC used for transferring information between kernel and user-space processes. It provides a full-duplex communication link between the two by way of standard socket APIs for user-space processes and a special kernel API for kernel modules. Netlink socket uses the address family AF_NETLINK, as compared to AF_INET used by TCP/IP socket. Each netlink socket feature defines its own protocol type in the kernel header file *include/linux/netlink.h* ."

<< Read more details from the above article; participants will find a simple netlink sockets demo userspace application and kernel module that communicate with each other within the provided courseware. >>

<< Source : "Linux Device Drivers" 3rd Ed, Rubini, Corbet, Hartman, O'Reilly publishers, Ch 9. >>

Using IO Memory

Despite the popularity of I/O ports in the x86 world, the main mechanism used to communicate with devices is through memory-mapped registers and device memory. Both are called I/O memory because the difference between registers and memory is transparent to software.

I/O memory is simply a region of RAM-like locations that the device makes available to the processor over the bus. This memory can be used for a number of purposes, such as holding video data or Ethernet packets, as well as implementing device registers that behave just like I/O ports (i.e., they have side effects associated with reading and writing them).

…

Depending on the computer platform and bus being used, I/O memory may or may not be accessed through page tables. When access passes though page tables, the kernel must first arrange for the physical address to be visible from your driver, and this usually means that you must call ioremap before doing any I/O. If no page tables are needed, I/O memory locations look pretty much like I/O ports, and you can just read and write to them using proper wrapper functions.

<<
Sidebar :: ioremap internally uses the vmalloc region of the kernel segment

[See the *ioremap* label on the right...]
Eg. on an ARM system:

```
# cat /proc/vmallocinfo
0xc8800000-0xc8802000    8192 smc_drv_probe+0x198/0x7c4 ioremap
0xc8802000-0xc8804000    8192 amba_kmi_probe+0xc4/0x15c ioremap
0xc8804000-0xc8806000    8192 amba_kmi_probe+0xc4/0x15c ioremap
0xc8832000-0xc8834000    8192 pl011_probe+0x88/0x198 ioremap
0xc8834000-0xc8836000    8192 pl011_probe+0x88/0x198 ioremap
0xc8836000-0xc8838000    8192 pl011_probe+0x88/0x198 ioremap
0xc8838000-0xc883a000    8192 pl011_probe+0x88/0x198 ioremap
```

```
0xc883a000-0xc8846000   49152 cramfs_uncompress_init+0x2c/0x60 pages=11 vmalloc
0xc8846000-0xc8889000  274432 jffs2_zlib_init+0x18/0xb4 pages=66 vmalloc
0xc8889000-0xc8895000   49152 jffs2_zlib_init+0x50/0xb4 pages=11 vmalloc
0xc8896000-0xc8898000    8192 clcdfb_probe+0x108/0x360 ioremap
#

>>
```

Whether or not ioremap is required to access I/O memory, direct use of pointers to I/O memory is discouraged. Even though (as introduced in the section “I/O Ports and I/O Memory”) I/O memory is addressed like normal RAM at hardware level, the extra care outlined in the section “I/O Registers and Conventional Memory” suggests avoiding normal pointers. The wrapper functions used to access I/O memory are safe on all platforms and are optimized away whenever straight pointer dereferencing can perform the operation.

Therefore, even though dereferencing a pointer works (for now) on the x86, failure to use the proper macros hinders the portability and readability of the driver.

--snip--

LINUX IO Memory | Quick Steps Summary

1 Port IO (PIO / Register IO)

- *Only present "physically" for the Intel x86 architecture*
  - LINUX abstracts the notion of I/O port-space for all architectures, even though other processors see "ports" simply as a mapped memory address
  - Occupies the low base address region: 0x0000XXXX, i.e., from 0x0 to 0x0000FFFF
  - Access routines:
    - **`request_region()`** **`[returns NULL on failure]`**
    - **`release_region()`**
  - I/O routines:
    - **`in[b|w|l]()`**
    - **`out[b|w|l]()`**

2 Memory Mapped IO (MMIO)

- The common way for drivers to access hardware – both registers as well as memory onboard a chip
- All access to these memory / register areas is via the Page Tables (virtual addresses)
- Includes ISA, PCI bus
  - Access routines:
    - **`request_mem_region()`**
    - **`release_mem_region()`**
  - I/O routines:
- **`ioremap() / ioremap_nocache()`** : performs the software mapping: it converts the hardware bus addresses to kernel virtual addresses so that read/write operations can be performed on the device.
  - Once ioremap() is done, the following memory functions can be used:
    - **`ioread[8|16|32](void *addr)`**
    - **`iowrite[8|16|32](u[8|16|32] value, void *addr)`**
    - **`memset_io(), memcpy_fromio(), memcpy_toio()`**
    - **`iounmap() (when done).`**

So, the typical sequence for MMIO:

1 request_mem_region
2 ioremap / ioremap_nocache
3 Actual I/O
  3.1 ioread[8|16|32]
  3.2 iowrite[8|16|32]

Some interesting wrappers around device resource management can be found in *lib/devres.c* . For example:

```
void __iomem *devm_request_and_ioremap(
            struct device *dev, struct resource *res);
```

<<

Example of using Port IO (PIO) in the mainline kernel

The i8042 chip is a keyboard and mouse controller, used on x86-based systems. The kernel driver is here: *drivers/input/serio/i8042-io.h*

```
...
  if (!request_region(I8042_DATA_REG, 16, "i8042"))
        return -EBUSY;
...
/*
 * Register numbers.
 */

#define I8042_COMMAND_REG   0x64
#define I8042_STATUS_REG    0x64
#define I8042_DATA_REG      0x60

static inline int i8042_read_data(void)
{
    return inb(I8042_DATA_REG);
}
static inline int i8042_read_status(void)
{
    return inb(I8042_STATUS_REG);
}
static inline void i8042_write_data(int val)
{
    outb(val, I8042_DATA_REG);
}
static inline void i8042_write_command(int val)
{
    outb(val, I8042_COMMAND_REG);
}
...
 release_region(I8042_DATA_REG, 16);
```

>>

Accessing I/O Memory - MMIO

On some platforms, you may get away with using the return value from ioremap as a pointer. Such use is not portable, and, increasingly, the kernel developers have been working to eliminate any such use. The proper way of getting at I/O memory is via a set of functions (defined via <asm/io.h>) provided for that purpose.

To read from I/O memory, use one of the following:

```
unsigned int ioread8(void *addr);
unsigned int ioread16(void *addr);
unsigned int ioread32(void *addr);
```

Here, addr should be an address obtained from ioremap (perhaps with an integer offset); the return value is what was read from the given I/O memory.

There is a similar set of functions for writing to I/O memory:

```
void iowrite8(u8 value, void *addr);
void iowrite16(u16 value, void *addr);
void iowrite32(u32 value, void *addr);
```

If you must read or write a series of values to a given I/O memory address, you can use the repeating versions of the functions:

```
void ioread8_rep(void *addr, void *buf, unsigned long count);
void ioread16_rep(void *addr, void *buf, unsigned long count);
void ioread32_rep(void *addr, void *buf, unsigned long count);
void iowrite8_rep(void *addr, const void *buf, unsigned long count);
void iowrite16_rep(void *addr, const void *buf, unsigned long count);
void iowrite32_rep(void *addr, const void *buf, unsigned long count);
```

These functions read or write count values from the given buf to the given addr. Note that count is expressed in the size of the data being written; ioread32_rep reads count 32-bit values starting at buf.

The functions described above perform all I/O to the given addr. If, instead, you need to operate on a block of I/O memory, you can use one of the following:

```
void memset_io(void *addr, u8 value, unsigned int count);
void memcpy_fromio(void *dest, void *source, unsigned int count);
void memcpy_toio(void *dest, void *source, unsigned int count);
```

These functions behave like their C library analogs.

If you read through the kernel source, you see many calls to an older set of functions when I/O memory is being used. These functions still work, but their use in new code is discouraged. Among other things, they are less safe because they do not perform the same sort of type checking. Nonetheless, we describe them here:

```
unsigned readb(address);
unsigned readw(address);
unsigned readl(address);
```

These macros are used to retrieve 8-bit, 16-bit, and 32-bit data values from I/O memory.

```
void writeb(unsigned value, address);
void writew(unsigned value, address);
void writel(unsigned value, address);
```

Like the previous functions, these functions (macros) are used to write 8-bit, 16-bit, and 32-bit data items.

Some 64-bit platforms also offer readq and writeq, for quad-word (8-byte) memory operations on the PCI bus. The quad-word nomenclature is a historical leftover from the times when all real processors had 16-bit words. Actually, the L naming used for 32-bit values has become incorrect too, but renaming everything would confuse things even more.

Examples of device drivers using IO Memory (MMIO)

I The device driver for TCG/TCPA TPM (trusted platform module) : drivers/char/tpm/tpm_tis.c :

```
#include <asm/io.h>   /* ioremap(),... */
...
void __iomem *iobase;       /* ioremapped address */
...

chip->vendor.iobase = ioremap(start, len);
…
/* Figure out the capabilities */
        intfcaps =
            ioread32(chip->vendor.iobase +
                     TPM_INTF_CAPS(chip->vendor.locality));
        dev_dbg(dev, "TPM interface capabilities (0x%x):\n",
                intfcaps);
…

for (i = 3; i < 16 && chip->vendor.irq == 0; i++) {
                        iowrite8(i, chip->vendor.iobase +
                                    TPM_INT_VECTOR(chip->vendor.locality));
…
```

II RealTek 8139cp Fast Ethernet network driver. From it's code (on 4.16.0 : drivers/net/8139cp.c):

```
#include <asm/io.h>   /* ioremap(),... */
...
#define cpr8(reg)       readb(cp->regs + (reg))
#define cpr16(reg)      readw(cp->regs + (reg))
#define cpr32(reg)      readl(cp->regs + (reg))
#define cpw8(reg,val)   writeb((val), cp->regs + (reg))
#define cpw16(reg,val)  writew((val), cp->regs + (reg))
#define cpw32(reg,val)  writel((val), cp->regs + (reg))
```

<< Notice how it's using the "older" IO routines above! >>

```
...
        resource_size_t pciaddr;
        void __iomem *regs;
...
        pciaddr = pci_resource_start(pdev, 1);    << from "bar 1" >>
```

```
...
        regs = ioremap(pciaddr, CP_REGS_SIZE);
      if (!regs) {
               rc = -EIO;
               dev_err(&pdev->dev, "Cannot map PCI MMIO (%Lx@%Lx)\n",
                       (unsigned long long)pci_resource_len(pdev, 1),
                      (unsigned long long)pciaddr);
               goto err_out_res;
      }
      cp->regs = regs;
        ...
```

*III **DEVMEM RW***

An opensource project (not in mainline), for read/write on any memory location on a PC or embedded board. This project clearly illustrates the use of the above ioremap, etc routines...

device-memory-readwrite
Read and/or Write to any memory location (RAM or H/W IO Memory) on a device.

Author: me :-)
Github: https://github.com/kaiwan/device-memory-readwrite

The README file:

Read and/or Write to pretty much any memory location (RAM or H/W IO Memory) on a device.

This project enables the user to read from and/or write to any generic memory location(s) on a device running the 2.6 / 3.x / 4.x Linux OS. This includes "regular" RAM as well as hardware IO Memory that's mapped into the kernel virtual address space. In fact, the driver will perform the mapping, given the hardware base address and length (as module parameters).

The read/write utility programs run in user-land and talk to the underlying kernel driver via the ioctl system call.

This could be extremely useful for driver authors, kernel developers, etc who want to peek/poke memory for prototyping purposes, learning, debug, testing, register lookups/writes and similar purposes.

To get started, please read clone the git tree, read the 'Devmem HOWTO.pdf' PDF

document, and get to it!

IV Writing device drivers in Linux: A brief tutorial

Note:

1. **MMIO Performance Issues** *

“MMIO is the most direct access to the card. A range of addresses is exposed to the CPU, where each write goes directly to the GPU <<or NIC or whatever>>. This allows the simplest form of communication of commands from the CPU to the GPU.

This type of programming is synchronous; writes are done by the CPU and executed on the GPU in a lockstep fashion. This leads to sub-par performance because each access turns into a packet on the bus and because the CPU has to wait for previous GPU commands to complete before submitting subsequent commands. For this reason MMIO is only used in the non-performance critical paths of today’s drivers.”

** Source: the open book: "Linux Graphics Drivers: an Introduction" by Stéphane Marchesin, Mar '12*

<<
So what's the high performance way of communicating with a peripheral device? The answer is DMA (Direct Memory Access). In fact, one can consider DMA operations to be broadly of two types:
- data transfer to/from CPU/device (i.e. the I/O part) over DMA
- command FIFO over DMA (where device “commands” are issued via DMA therefore becoming asynchronous to the processor and thus much faster).
>>

2. In a driver, when performing an operation on a register (often, set/clear a bit(s)), it is important to use this sequence:

Read
Modify
Write

When doing this, take a (spin)lock (as a hardware register is typically a shared resource

on the system).

Eg.

```
...
u32 val;

spin_lock(&mlock);
/* Enable MYXXX counter */
val = readl(XXX_TIMER_ENABLE_REG);
val |= 0x5;
writel(val, XXX_TIMER_ENABLE_REG);
spin_unlock(&mlock);
…
```

Alternatively, easier, recommended, use the bit-wise atomic operators (*set_bit(), clear_bit()*, etc).

3. Using the 'volatile' keyword:

Read this article "How to Use C's volatile Keyword".

However, a word of caution from the kernel folk: *Documentation/volatile-considered-harmful.txt !*

Appendix A :: Testing the *access_ok()* and *copy_to_user()* using an application and device driver

A device driver's read method essentially works by reading the number of bytes requested from the hardware (or whatever) and copying that data to user-space by invoking the *copy_to_user()* macro.

That's fine, but what if the user application is malicious (or buggy) and intentionally (or otherwise) passes an arbitrary (virtual) address within the process address space; this could potentially be anything – a part of a library's text or data, or the application process'es data or stack segment.

What happens then? Lets test this.
Below, the 'C' application is designed to parse the starting virtual address of it's own "virtual segments" (technically, it's VMAs; using proc), offset it by a small amount and pass that address to the driver's read method. The device driver will now attempt to write to that address!

Sample Session

<< Participants can find the source of this driver on the provided CD >>

```
# insmod badcdrv.ko
# dmesg
[23659.596329] badcdrv: registered with major number 250
[23659.596339] badcdrv: cdev badcdrv.0 added
[23659.596474]  badcdrv: Device node /dev/badcdrv.0 created.
# ls -l /dev/badcdrv.0
crw------- 1 root root 250, 0 2011-03-02 15:42 /dev/badcdrv.0
# ./app_badread
Usage: ./app_badread device_file num_bytes_to_read
#
# ./app_badread /dev/badcdrv.0 100
./app_badread: PID 31855.

VMA: 0011b000-00272000 r-xp 00000000 08:0a 1065877    /lib/libc-
2.12.1.so
```

```
sAddr = 0011b000 dest_addr = 0x0011b500
read failed: Input/output error       << as expected >>

VMA: 00272000-00274000 r--p 00157000 08:0a 1065877    /lib/libc-
2.12.1.so
sAddr = 00272000 dest_addr = 0x00272500
read failed: Input/output error

VMA: 00274000-00275000 rw-p 00159000 08:0a 1065877    /lib/libc-
2.12.1.so
sAddr = 00274000 dest_addr = 0x00274500
Segmentation fault (core dumped)     << whoops! >>
# dmesg
[23659.596329] badcdrv: registered with major number 250
[23659.596339] badcdrv: cdev badcdrv.0 added
[23659.596474]  badcdrv: Device node /dev/badcdrv.0 created.
[23690.622709] badcdrv:cdrv_open:88: Device node with minor # 0 being
used
[23690.628211] badcdrv:badread:46:
[23690.628214] -----PID 31855. userspace dest addr = 0x0011b500
[23690.628221] badcdrv:badread:62:  access_ok check passed.
[23690.628230] badcdrv:badread:copy_to_user failed! Ret=100
                                                  << as expected >>
[23690.628349] badcdrv:badread:46:
[23690.628352] -----PID 31855. userspace dest addr = 0x00272500
[23690.628357] badcdrv:badread:62:  access_ok check passed.
[23690.628364] badcdrv:badread:copy_to_user failed! Ret=100
                                                  << as expected >>
[23690.628426] badcdrv:badread:46:
[23690.628428] -----PID 31855. userspace dest addr = 0x00274500
[23690.628434] badcdrv:badread:62:  access_ok check passed.
[23690.628439] badcdrv:badread:70:  copy_to_user to 0x00274500
succeeded!
                                                  << not expected >>
[23690.628450] app_badread[31855]: segfault at 8 ip 001583d1 sp bfb6740c
error 4 in libc-2.12.1.so[11b000+157000]
#
```

We find that the *access_ok()* method really makes no difference – it merely disallows access to virtual addresses above PAGE_OFFSET. However, we find that the *copy_to_user()* macro does work – but only sometimes! - disallowing the attempted write.

This experiment underscores the need for the driver author to validate user-space addresses before performing any I/O to them.

In this trial, if one thinks about it, it's not the driver's fault, it's really the application's for providing illegal addresses!

Appendix B :: Using the Proc Filesystem ABI

<<
Warning!!
Procfs for anything other than in-kernel use is likely to soon be (or already is!) deprecated!
*We **Strongly** suggest you use other interfaces (like debugfs, sysfs or netlink sockets) in place of procfs for driver-based interfaces.*
>>

Creating our own entry in /proc

Some device drivers also export information via /proc, and yours can do so as well. The */proc* filesystem is dynamic, so your module can add or remove entries at any time.

Fully featured */proc* entries can be complicated beasts; among other things, they can be written to as well as read from. Most of the time, however, */proc* entries are read-only files. This section will concern itself with the simple read-only case. Those who are interested in implementing something more complicated can look here for the basics; the kernel source may then be consulted for the full picture.

All modules that work with /proc should include ***<linux/proc_fs.h>*** to define the proper functions.

Typically, using */proc* in your driver involves the following three steps:

1. Creating your /proc file(s):

There are two ways of setting up your /proc entries, depending on what versions of the kernel you wish to support. The easiest method, only available in the 2.4 kernel upward is to simply call `create_proc_read_entry` .

```
create_proc_read_entry("test",
                       0440,    /* default mode (perms) */
                       NULL, /* parent dir */
                       test_read_procmem,
                       NULL /* client data */);
```

```
                    --OR--

struct proc_dir_entry* create_proc_entry(
                                const char* name,
                                mode_t mode,
                                struct proc_dir_entry* parent);
```

This function creates a regular file with the name *name*, file mode *mode* in the directory *parent*. To create a file in the root of the procfs (i.e. in /proc), use NULL as parent parameter. When successful, the function will return a pointer to the freshly created struct proc_dir_entry; otherwise it will return NULL.

Note that it is specifically supported that you can pass a path that spans multiple directories. For example *create_proc_entry("drivers/via0/info")* will create the via0 directory if necessary, with standard 0755 permissions.

If you only want to be able to read the file, the function *create_proc_read_entry* (shown previous to this one) may be used to create and initialise the procfs entry in one single call.

For example:

```
static struct proc_dir_entry *res;
...

<< step 1. in the init_module function >>
/* create the /proc entry */
if( (res=create_proc_entry("driver/test", 0664, NULL)) == NULL ) {
      printk(KERN_DEBUG "proc entry failed\n");
      return -ENOMEM;
}
```

2. Setup the read/write "hooks"

Once the /proc entries are created, initialize the returned struct members appropriately, to setup the read and/or write "hook" or callback functions (in your driver).

```
<< step 2. continuing the above example code, >>
...
res->read_proc = test_read_proc; /* ptr to function, the
                                    read callback */
res->write_proc = test_write_proc; /* ptr to function, the
                                      write callback */
res->data=NULL;                      /* "client" data, to multiplex
...                                     between several /proc
```

```
                                        entries
                                       */
```

To create a read-only /proc file, **your driver must implement a function to produce the data when the file is read** (test_read_proc, in the above example). When some process reads the file (using the read system call), the request will reach your module by means of one of two different interfaces, according to what you registered. We'll leave registration for later in this section and jump directly to the description of the reading interfaces.

In both cases the kernel allocates a page of memory (i.e., PAGE_SIZE bytes) where the driver can write data to be returned to user space.

The recommended interface is read_proc, but an older interface named get_info also exists.

```
int (*read_proc)(char *page, char **start, off_t offset, int count, int *eof,
void *data);
```

The page pointer is the buffer **where you'll write** your data - note that it is *not* necessary to use the copy_to_user macro- page is already a kernel virtual address; start is used by the function to say where the interesting data has been written in page; offset and count have the same meaning as in the read implementation. The **eof** argument points to an integer that **must be set** by the driver to signal that it has no more data to return, while data is a driver-specific data pointer you can use for internal bookkeeping.

The main problem with the original implementation of user extensions to the /proc filesystem was use of a **single memory page for data transfer**. This limited the total size of a user file to the page size- 4 KB (or whatever was appropriate for the host platform). The start argument is there to implement large data files, but it can be ignored (not really used on modern kernels).

In the above example, when a users-space process reads (by using cat or whatever), the file /proc/driver/test, **the kernel VFS layer will *call back* the function test_read_proc**. Similarly, when userspace writes into /proc/driver/test, the kernel VFS will *call back* the function test_write_proc. The functions implement what's necessary (they're often used to help debug the driver as well).

The signature of the **“write” callback** function is simpler:

```
int (*write_proc)(struct file *filp, const char __user * buffer, unsigned long
count, void * data)
```

buffer, of course, is the user-space buffer to be populated by this function, for *count* bytes.

<< step 3. again continuing the above example code, >>

```
...
static int test_write_proc(struct file *filp, const char * buffer,
                           unsigned long count, void * data)
{
       /*
      * Write whatever is passed (in buffer, for count
      * bytes) as necessary into the driver.
        * Note that it is necessary here to use the
        * copy_from_user() macro to transfer the user -space
        * data buffer to kernel space.
      */
       copy_from_user(....);
       ....
       return count;   // amount of data actually written
}
```

4. Removing the /proc entry

Also, in the cleanup routine, you *must* ensure that the /proc entry created is **removed**. This is done by:

```
       ...
       remove_proc_entry("driver/test", NULL);
       ...
```

Important Notes

A limitation of using /proc – based entries is that, when a user-space process reads from a proc file, the maximum amount of data that can be passed back by the kernel/driver to it (in one go) is limited to one page. For most drivers this is more than sufficient and hence poses no serious issue. There may be cases, however, when you do want to export a large amount of information (more than one page) to user-space. In situations like this, the kernel supports a cleaned-up interface called seq_file. The seq_file interface assumes that you are creating a virtual file that steps through a sequence of items that must be returned to user-space. To use seq-file, you must create an "iterator" object that can establish a position within the sequence, step forward, and output one item in the sequence.
For further details and a usage example, see Chapter 4 "Debugging Techniques", LDD3*.

Online Resources:
A SeqFile Tutorial (from the Linux Kernel Module Programming Guide)
Driver porting: The seq_file interface – LWN.net article
KernelNewbies SeqFileHowTo

<< End Document >>