Introduction to Loadable Kernel Modules

What Is A Kernel Module?

So, you want to write a kernel module. You know C, you've written a few normal programs to run as processes, and now you want to get to where the real action is, to where a single wild pointer can wipe out your file system and a core dump means a reboot. What exactly is a kernel module? Modules are pieces of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system. For example, one type of module is the device driver, which allows the kernel to access hardware connected to the system. Without modules, we would have to build monolithic kernels and add new functionality directly into the kernel image. Besides having larger kernels, this has the disadvantage of requiring us to rebuild and reboot the kernel every time we want new functionality.

How Do Modules Get Into The Kernel?

You can see what modules are already loaded into the kernel by running lsmod, which gets its information by reading the file /proc/modules. How do these modules find their way into the kernel? When the kernel needs a feature that is not resident in the kernel, the kernel module daemon execs modprobe to load the module in. modprobe is passed a string in one of two forms:

- A module name like ny_module or ppp.
- A more generic identifier like char-major-10-30.

If modprobe is handed a generic identifier, it first looks for that string in the file /etc/modules.conf.

If it finds an alias line like:

```
alias char-major-10-30 softdog
```

it knows that the generic identifier refers to the module softdog.o.

Linux distros provide modprobe, insmod and depmod as a package called modutils or mod-utils.

Before We Begin

Before we delve into code, there are a few issues we need to cover. Everyone's system is different and everyone has their own groove. Getting your first "hello world" program to compile and load correctly can sometimes be a trick. Rest assured, after you get over the initial hurdle of doing it for the first time, it will be smooth sailing thereafter.

Modversioning

A module compiled for one kernel won't load if you boot a different kernel unless you enable CONFIG_MODVERSIONS in the kernel. We won't go into module versioning until later in this guide. Until we cover modversions, the examples in the guide may not work if you're running a kernel with modversioning turned on. However, most stock Linux distro kernels come with it turned on. If you're having trouble loading the modules because of versioning errors, compile a kernel with modversioning turned off.

Compiling Issues and Kernel Version

Very often, Linux distros will distribute kernel source that has been patched in various non-standard ways, which may cause trouble.

A more common problem is that some Linux distros distribute incomplete kernel headers. You'll need to compile your code using various header files from the Linux kernel. Murphy's Law states that the headers that are missing are exactly the ones that you'll need for your module work.

To avoid these two problems, I highly recommend that you download, compile and boot into a fresh, stock Linux kernel which can be downloaded from any of the Linux kernel mirror sites. See the Linux Kernel HOWTO for more details. Ironically, this can also cause a problem. By default, gcc on your system may look for the kernel headers in their default location rather than where you installed the new copy of the kernel (usually in /usr/src/. This can be fixed by using gcc's -I switch.

Introduction to Loadable Kernel Modules

A Simplest Module

Here's the simplest module possible. Don't compile it yet; we'll cover module compilation in the next section.

Example1. hello-1.c

```
/* hello-1.c - The simplest kernel module.   */
#include <linux/module.h>                    /* Needed by all modules */
#include <linux/kernel.h>                    /* Needed for KERN_ALERT */
int init_module(void)
{
     printk("<1>Hello world 1.\n");
     // A non 0 return means init_module failed; module can't be loaded.
     return 0;
}
void cleanup_module(void)
{
    printk(KERN_ALERT "Goodbye world 1.\n");
}
```

Kernel modules must have at least two functions: a "start" (initialization) function called init_module() which is called when the module is insmoded into the kernel, and an "end" (cleanup) function called cleanup_module() which is called just before it is rmmoded. Actually, things have changed starting with kernel 2.3.13. You can now use whatever name you like for the start and end functions of a module. In

fact, the new method is the preferred method. However, many people still use init_module() and cleanup_module() for their start and end functions.

Typically, init_module() either registers a handler for something with the kernel, or it replaces one of the kernel functions with its own code. The cleanup_module() function is supposed to undo whatever init_module() did, so the module can be unloaded safely.

Lastly, every kernel module needs to include linux/module.h. We needed to include linux/kernel.h only for the macro expansion for the printk() log level, KERN_ALERT.

Introducing printk()

Despite what you might think, printk() was not meant to communicate information to the user, even though we used it for exactly this purpose in hello-1! It happens to be a logging mechanism for the kernel, and is used to log information or give warnings. Therefore, each printk() statement comes with a priority, which is the <1> and KERN_ALERT you see. There are 8 priorities and the kernel has macros for them, so you don't have to use cryptic numbers, and you can view them in linux/kernel.h. If you don't specify a priority level, the default priority, DEFAULT_MESSAGE_LOGLEVEL, will be used. In practise, don't use number, like <4>. Always use the macro, like KERN_WARNING. If the priority is less than int console_loglevel, the message is printed on your current terminal. If both syslogd and klogd are running, then the message will also get appended to /var/log/messages, whether it got printed to the console or not. We use a high priority, like KERN_ALERT, to make sure the printk() messages get printed to your console rather than just logged to your logfile. When you write real modules, you'll want to use priorities that are meaningful for the situation at hand.

Compiling Kernel Modules

Kernel modules need to be compiled with certain gcc options to make them work. In addition, they also need to be compiled with certain symbols defined. This is because the kernel header files need to behave differently, depending on whether we're compiling a kernel module or an executable. You can define symbols using gcc's -D option, or with the #define preprocessor command.

- -c: A kernel module is not an independant executable, but an object file which will be linked into the kernel during runtime using insmod. As a result, modules should be compiled with the -c flag.
- -O2: The kernel makes extensive use of inline functions, so modules must be compiled with the optimization flag turned on. Without optimization, some of the assembler macros calls will be mistaken by the compiler for function calls. This will cause loading the module to fail, since insmod won't find those functions in the kernel.
- -W -Wall: A programming mistake can take take your system down. You should always turn on compiler warnings, and this applies to all your compiling endeavors, not just module compilation.
- -isystem /lib/modules/`uname -r`/build/include: You must use the kernel headers of the kernel you're compiling against. Using the default /usr/include/linux won't work.
- -D__KERNEL__: Defining this symbol tells the header files that the code will be run in kernel mode, not as a user process.
- -DMODULE: This symbol tells the header files to give the appropriate definitions for a kernel module. We use gcc's -isystem option instead of -I because it tells gcc to surpress some "unused variable" warnings that -W -Wall causes when you include module.h. By using -isystem under gcc-3.0, the kernel header files are treated specially, and

the warnings are surpressed. If you instead use -I (or even -isystem under gcc 2.9x), the "unused variable" warnings will be printed. Just ignore them if they do.

So, let's look at a simple Makefile for compiling a module named hello-1.c:

Example 2. Makefile for a basic kernel module

```
TARGET      :=   hello-1
WARN        :=   -W -Wall -Wstrict-prototypes -Wmissing-prototypes
INCLUDE     :=   -isystem /lib/modules/`uname -r`/build/include
CFLAGS      :=   -O2 -DMODULE -D__KERNEL__ ${WARN} ${INCLUDE}
CC          :=   gcc-3.0
${TARGET}.o: ${TARGET}.c
.PHONY: clean
clean:
     rm -rf {TARGET}.o
```

As an exercise to the reader, compile hello-1.c and insert it into the kernel with insmod ./hello-1.o. Neat, eh? All modules loaded into the kernel are listed in /proc/modules. Go ahead and cat that file to see that your module is really a part of the kernel. Congratulations, you are now the author of Linux kernel code! When the novelty wares off, remove your module from the kernel by using rmmod hello-1. Take a look at /var/log/messages just to see that it got logged to your system logfile.

Here's another exercise to the reader. See that comment above the return statement in init_module()?

Change the return value to something non-zero, recompile and load the module again. What happens?

Example-3. Hello World (part 2)

As of Linux 2.4, you can rename the init and cleanup functions of your modules; they no longer have to be called init_module() and cleanup_module() respectively. This is done with the module_init() and module_exit() macros. These macros are defined in linux/init.h. The only caveat is that your init and cleanup functions must be defined before calling the macros, otherwise you'll get compilation errors.

Here's an example of this technique:

Example 3. hello-2.c

```
#include <linux/module.h>          // Needed by all modules
#include <linux/kernel.h>          // Needed for KERN_ALERT
#include <linux/init.h>              // Needed for the macros
static int hello_2_init(void)
{
     printk(KERN_ALERT "Hello, world 2\n");
     return 0;
}
static void hello_2_exit(void)
{
```

```
    printk(KERN_ALERT "Goodbye, world 2\n");
}
module_init(hello_2_init);
module_exit(hello_2_exit);
```

Here's a more advanced Makefile which will compile both our modules at the same time. It's optimized for brevity and scalability.

Example 2-4. Makefile for both our modules

```
WARN        := -W -Wall -Wstrict-prototypes -Wmissing-prototypes
INCLUDE     := -isystem /lib/modules/`uname -r`/build/include
CFLAGS      := -O2 -DMODULE -D__KERNEL__ ${WARN} ${INCLUDE}
CC          := gcc-3.0
OBJS        := ${patsubst %.c, %.o, ${wildcard *.c}}
all: ${OBJS}
.PHONY: clean
clean:
    rm -rf *.o
```

Hello World (part 3): The __init and __exit Macros

The __init macro causes the init function to be discarded and its memory freed once the init function finishes for built-in drivers, but not loadable modules. If you think about when the init function is invoked, this makes perfect sense. There is also an __initdata which works similarly to __init but for init variables rather than functions. The __exit macro causes the omission of the function when the module is built into the kernel, and like __exit, has no effect for loadable modules. Again, if you consider

when the cleanup function runs, this makes complete sense; built-in drivers don't need a cleanup function, while loadable modules do. These macros are defined in linux/init.h and serve to free up kernel memory.

Example 5. hello-3.c

```
#include <linux/module.h>          /* Needed    by all modules */
#include <linux/kernel.h>          /* Needed    for KERN_ALERT */
#include <linux/init.h>            /* Needed    for the macros */
static int hello3_data __initdata = 3;
static int __init hello_3_init(void)
{
    printk(KERN_ALERT "Hello, world %d\n", hello3_data);
    return 0;
}
static void __exit hello_3_exit(void)
{
    printk(KERN_ALERT "Goodbye, world 3\n");
}
module_init(hello_3_init);
module_exit(hello_3_exit);
```

Hello World (part 4): Licensing and Module Documentation

If you're running kernel 2.4 or later, you might have noticed something like this when you loaded the previous example modules:

```
# insmod hello-3.o
Warning: loading hello-3.o will taint the kernel: no license
    See http://www.tux.org/lkml/#export-tainted for information about tainted modules Hello, world 3
Module hello-3 loaded, with warnings
```

In kernel 2.4 and later, a mechanism was devised to identify code licensed under the GPL (and friends) so people can be warned that the code is non open-source. This is accomplished by the MODULE_LICENSE() macro which is demonstrated in the next piece of code. By setting the license to GPL, you can keep the warning from being printed. This license mechanism is defined and documented in linux/module.h. Similarly, MODULE_DESCRIPTION() is used to describe what the module does, MODULE_AUTHOR() declares the module's author, and MODULE_SUPPORTED_DEVICE() declares what types of devices the module supports. These macros are all defined in linux/module.h and aren't used by the kernel itself. They're simply for documentation and can be viewed by a tool like objdump.

Example 6. hello-4.c

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#define DRIVER_AUTHOR "Peiter Jay Salzman <p@dirac.org>"
#define DRIVER_DESC        "A sample driver"
int init_hello_3(void);
```



```
void cleanup_hello_3(void);
static int init_hello_4(void)
{
    printk(KERN_ALERT "Hello, world 4\n");
    return 0;
}
static void cleanup_hello_4(void)
{
    printk(KERN_ALERT "Goodbye, world 4\n");
}
module_init(init_hello_4);
module_exit(cleanup_hello_4);
MODULE_LICENSE("GPL");
    // Get rid of taint message by declaring code as GPL.
    /* Or with defines, like this:   */
MODULE_AUTHOR(DRIVER_AUTHOR);
    // Who wrote this module?
MODULE_DESCRIPTION(DRIVER_DESC);
    // What does this module do?
/* This module uses /dev/testdevice. The MODULE_SUPPORTED_DEVICE macro might be used in the future to help automatic configuration of modules, but is currently unused other than for documentation purposes.
*/
MODULE_SUPPORTED_DEVICE("testdevice");
```

Hello Kernel

Passing Command Line Arguments to a Module

Modules can take command line arguments, but not with the argc/argv you might be used to. To allow arguments to be passed to your module, declare the variables that will take the values of the command line arguments as global and then use the MODULE_PARM() macro, (defined in

linux/module.h) to set the mechanism up. At runtime, insmod will fill the variables with any command line arguments that are given. The variable declarations and macros should be placed at the beginning of the module for clarity. The example code should clear up my admittedly lousy explanation.

The MODULE_PARM() macro takes 2 arguments: the name of the variable and its type. The supported variable types are "b": single byte, "h": short int, "i": integer, "l": long int and "s": string. Strings should be declared as "char *" and insmod will allocate memory for them. You should always try to give the variables an initial default value. This is kernel code, and you should program defensively. For example:

```
int myint = 3;

char *mystr;

MODULE_PARM (myint, "i");

MODULE_PARM (mystr, "s");
```

Arrays are supported too. An integer value preceding the type in MODULE_PARM will indicate an array of some maximum length. Two numbers separated by a '-' will give the minimum and maximum number of values. For example, an array of shorts with at least 2 and no more than 4 values could be declared as:

```
int myshortArray[4];

MODULE_PARM (myintArray, "2-4i");
```

A good use for this is to have the module variable's default values set, like which IO port or IO memory to use. If the variables contain the default values, then perform autodetection. Otherwise, keep the current value. This will be made clear later on. For now, I just want to demonstrate passing arguments to a module.

Example 1. hello1.c

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Pravjot Singh");
static    short int myshort = 1;
static    int myint = 420;
static    long int mylong = 9999;
static    char *mystring = "blah";
MODULE_PARM       (myshort, "h");
MODULE_PARM       (myint, "i");
MODULE_PARM       (mylong, "l");
MODULE_PARM       (mystring, "s");
static int __init hello_1_init(void)
{
     printk(KERN_ALERT "Hello, world 1\n==============\n");
     printk(KERN_ALERT "myshort is a short integer: %hd\n", myshort);
     printk(KERN_ALERT "myint is an integer: %d\n", myint);
     printk(KERN_ALERT "mylong is a long integer: %ld\n", mylong);
     printk(KERN_ALERT "mystring is a string: %s\n", mystring);
     return 0;
```



```
}
static void __exit hello_1_exit(void)
{
    printk(KERN_ALERT "Goodbye, world 1\n");
}
module_init(hello_1_init);
module_exit(hello_1_exit);
```

Supercalifragilisticexpialidocious

Modules Spanning Multiple Files

Sometimes it makes sense to divide a kernel module between several source files. In this case, you need to:

1. In all the source files but one, add the line #define __NO_VERSION__. This is important because module.h normally includes the definition of kernel_version, a global variable with the kernel version the module is compiled for. If you need version.h, you need to include it yourself, because module.h won't do it for you with __NO_VERSION__.

2. Compile all the source files as usual.

3. Combine all the object files into a single one. Under x86, use ld -m elf_i386 -r -o <module name.o> <1st src file.o> <2nd src file.o>.

Here's an example of such a kernel module.

Example 2. start.c

```
#include <linux/kernel.h>               /* We're doing kernel work */
#include <linux/module.h>               /* Specifically, a module */
int init_module(void)
{
  printk("Hello, world - this is the kernel speaking\n");
  return 0;
}
```

The next file:

Example 3. stop.c

```
#if defined(CONFIG_MODVERSIONS) && ! defined(MODVERSIONS)
    #include <linux/modversions.h>      /* Will be explained later */
    #define MODVERSIONS
#endif
#include <linux/kernel.h>               /* We're doing kernel work */
#include <linux/module.h>               /* Specifically, a module */
#define __NO_VERSION__    /* It's not THE file of the kernel module */
#include <linux/version.h>
        /* Not included by module.h because of __NO_VERSION__ */
void cleanup_module()
{
   printk("<1>Short is the life of a kernel module\n");
}
```


And finally, the makefile:

Example 4. Makefile for a multi-filed module

```
CC=gcc
MODCFLAGS := -O -Wall -DMODULE -D__KERNEL__
hello.o:            hello2_start.o hello2_stop.o
   ld -m elf_i386 -r -o hello2.o hello2_start.o hello2_stop.o
start.o: hello2_start.c
   ${CC} ${MODCFLAGS} -c hello2_start.c
stop.o: hello2_stop.c
   ${CC} ${MODCFLAGS} -c hello2_stop.c
```

Preliminaries

Modules vs Programs

1. How modules begin and end

A program usually begins with a main() function, executes a bunch of instructions and terminates upon completion of those instructions. Kernel modules work a bit differently. A module always begin with either the init_module or the function you specify with module_init call. This is the entry function for modules; it tells the kernel what functionality the module provides and sets up the kernel to run the module's functions when they're needed. Once it does this, entry function returns and the module does nothing until the kernel wants to do something with the code that the module provides. All modules end by calling either cleanup_module or the function you specify with the module_exit call. This is the exit function for modules; it undoes whatever entry function did. It unregisters the functionality that the entry function registered. Every module must have an entry function and an exit function. Since there's more than one way to specify entry and exit functions.

2. Functions available to modules

Programmers use functions they don't define all the time. A prime example of this is printf(). You use these library functions which are provided by the standard C library, libc. The definitions for these functions don't actually enter your program until the linking stage, which insures that the code (for printf() for example) is available, and fixes the call instruction to point to that code. Kernel modules are different here, too. In the hello world example, you might have noticed that we used a function, printk() but didn't include a standard I/O library. That's because modules are object files whose symbols get resolved upon insmod'ing. The definition for the symbols comes from the kernel itself; the only external functions you can use are the ones provided by the kernel. If you're curious about what symbols have been exported by your kernel, take a

look at /proc/ksyms. One point to keep in mind is the difference between library functions and system calls. Library functions are higher level, run completely in user space and provide a more convenient interface for the programmer to the functions that do the real work---system calls. System calls run in kernel mode on the user's behalf and are provided by the kernel itself. The library function printf() may look like a very general printing function, but all it really does is format the data into strings and write the string data using the low-level system call write(), which then sends the data to standard output.

Would you like to see what system calls are made by printf()? It's easy! Compile the following program:

```
#include <stdio.h>

int main(void)

{ printf("hello"); return 0; }
```

with gcc -Wall -o hello hello.c. Run the exectable with strace hello. Are you impressed? Every line you see corresponds to a system call. strace is a handy program that gives you details about what system calls a program is making, including which call is made, what its arguments are what it returns. It's an invaluable tool for figuring out things like what files a program is trying to access. Towards the end, you'll see a line which looks like write(1, "hello", 5hello). There it is. The face behind the printf() mask. You may not be familiar with write, since most people use library functions for file I/O (like fopen, fputs, fclose). If that's the case, try looking at man 2 write. The 2nd man section is devoted to system calls (like kill() and read(). The 3rd man section is devoted to library calls, which you would probably be more familiar with (like cosh() and random()).

You can even write modules to replace the kernel's system calls, which we'll do shortly. Crackers often make use of this sort of thing for

backdoors or trojans, but you can write your own modules to do more benign things, like have the kernel write Tee hee, that tickles! everytime someone tries to delete a file on your system.

3. User Space vs Kernel Space

A kernel is all about access to resources, whether the resource in question happens to be a video card, a hard drive or even memory. Programs often compete for the same resource. As I just saved this document, updatedb started updating the locate database. My vim session and updatedb are both using the hard drive concurrently. The kernel needs to keep things orderly, and not give users access to resources whenever they feel like it. To this end, a CPU can run in different modes. Each mode gives a different level of freedom to do what you want on the system. The Intel 80386 architecture has 4 of these modes, which are called rings. Unix uses only two rings; the highest ring (ring 0, also known as `supervisor mode' where everything is allowed to happen) and the lowest ring, which is called `user mode'. Typically, you use a library function in user mode. The library function calls one or more system calls, and these system calls execute on the library function's behalf, but do so in supervisor mode since they are part of the kernel itself. Once the system call completes its task, it returns and execution gets transfered back to user mode.

4. Name Space

When you write a small C program, you use variables which are convenient and make sense to the reader. If, on the other hand, you're writing routines which will be part of a bigger problem, any global variables you have are part of a community of other peoples' global variables; some of the variable names can clash. When a program has lots of global variables which aren't meaningful enough to be distinguished, you get namespace pollution. In large projects, effort must be made to

remember reserved names, and to find ways to develop a scheme for naming unique variable names and symbols. When writing kernel code, even the smallest module will be linked against the entire kernel, so this is definitely an issue. The best way to deal with this is to declare all your variables as static and to use a well-defined prefix for your symbols. By convention, all kernel prefixes are lowercase. If you don't want to declare everything as static, another option is to declare a symbol table and register it with a kernel. The file /proc/ksyms holds all the symbols that the kernel knows about and which are therefore accessible to your modules since they share the kernel's codespace.

5. Code space

Memory management is a very complicated subject---1. If you haven't thought about what a segfault really means, you may be surprised to hear that pointers don't actually point to memory locations. Not real ones, anyway. When a process is created, the kernel sets aside a portion of real physical memory and hands it to the process to use for its executing code, variables, stack, heap and other things which a computer scientist would know about. This memory begins with 0 and extends up to whatever it needs to be. Since the memory space for any two processes don't overlap, every process that can access a memory address, say 0xbffff978, would be accessing a different location in real physical memory! The processes would be accessing an index named 0xbffff978 which points to some kind of offset into the region of memory set aside for that particular process. For the most part, a process like our Hello, World program can't access the space of another process, although there are ways which we'll talk about later.

The kernel has its own space of memory as well. Since a module is code which can be dynamically inserted and removed in the kernel, it shares the kernel's codespace rather than having its own. Therefore, if your module segfaults, the kernel segfaults. And if you start writing over data

because of an off-by-one error, then you're trampling on kernel code. This is even worse than it sounds, so try your best to be careful. By the way, I would like to point out that the above discussion is true for any operating system which uses a monolithic kernel. There are things called microkernels which have modules which get their own codespace. The GNU Hurd and QNX Neutrino are two examples of a microkernel.

6. Device Drivers

One class of module is the device driver, which provides functionality for hardware like a TV card or a serial port. On unix, each piece of hardware is represented by a file located in /dev named a device file which provides the means to communicate with the hardware. The device driver provides the communication on behalf of a user program. So the es1370.o sound card device driver might connect the /dev/sound device file to the Ensoniq IS1370 sound card. A userspace program like mp3blaster can use /dev/sound without ever knowing what kind of sound card is installed.

6.1. Major and Minor Numbers

Let's look at some device files. Here are device files which represent the first three partitions on the primary master IDE hard drive:

```
# ls -l /dev/hda[1-3]
brw-rw---- 1 root disk     3, 1 Jul  5  2000 /dev/hda1
brw-rw---- 1 root disk     3, 2 Jul  5  2000 /dev/hda2
brw-rw---- 1 root disk     3, 3 Jul  5  2000 /dev/hda3
```

Notice the column of numbers separated by a comma? The first number is called the device's major number. The second number is the minor number. The major number tells you which driver is used to access the hardware. Each driver is assigned a unique major number; all device files

with the same major number are controlled by the same driver. All the above major numbers are 3, because they're all controlled by the same driver.

The minor number is used by the driver to distinguish between the various hardware it controls. Returning to the example above, although all three devices are handled by the same driver they have unique minor numbers because the driver sees them as being different pieces of hardware.

Devices are divided into two types: character devices and block devices. The difference is that block devices have a buffer for requests, so they can choose the best order in which to respond to the requests. This is important in the case of storage devices, where it's faster to read or write sectors which are close to each other, rather than those which are further apart. Another difference is that block devices can only accept input and return output in blocks, whereas character devices are allowed to use as many or as few bytes as they like. Most devices in the world are character, because they don't need this type of buffering, and they don't operate with a fixed block size. You can tell whether a device file is for a block device or a character device by looking at the first character in the output of ls –l. If it's `b' then it's a block device, and if it's `c' then it's a character device. The devices you see above are block devices. Here are some character devices (the serial ports):

```
crw-rw----    1 root     dial      4, 64 Feb 18 23:34 /dev/ttyS0
crw-r-----    1 root     dial      4, 65 Nov 17 10:26 /dev/ttyS1
crw-rw----    1 root     dial      4, 66 Jul  5  2000 /dev/ttyS2
crw-rw----    1 root     dial      4, 67 Jul  5  2000 /dev/ttyS3
```

If you want to see which major numbers have been assigned, you can look at /usr/src/linux/Documentation/devices.txt. When the system was

installed, all of those device files were created by the mknod command. To create a new char device named `coffee' with major/minor number 12 and 2, simply do mknod /dev/coffee c 12 2. You don't have to put your device files into /dev, but it's done by convention. Linus put his device files in /dev, and so should you. However, when creating a device file for testing purposes, it's probably OK to place it in your working directory where you compile the kernel module. Just be sure to put it in the right place when you're done writing the device driver. I would like to make a few last points which are implicit from the above discussion, but I'd like to make them explicit just in case. When a device file is accessed, the kernel uses the major number of the file to determine which driver should be used to handle the access. This means that the kernel doesn't really need to use or even know about the minor number. The driver itself is the only thing that cares about the minor number. It uses the minor number to distinguish between different pieces of hardware. By the way, when I say `hardware', I mean something a bit more abstract than a PCI card that you can hold in your hand. Look at these two device files:

```
% ls –l /dev/fd0 /dev/fd0u1680
brwxrwxrwx  1  root     floppy     2,   0  Jul  5  2000      /dev/fd0
brw-rw----1  root     floppy     2,  44  Jul  5  2000/dev/fd0u1680
```

By now you can look at these two device files and know instantly that they are block devices and are handled by same driver (block major 2). You might even be aware that these both represent your floppy drive, even if you only have one floppy drive. Why two files? One represents the floppy drive with 1.44 MB of storage. The other is the same floppy drive with 1.68 MB of storage, and corresponds to what some people call a `superformatted' disk. One that holds more data than a standard formatted floppy. So here's a case where two device files with different minor number actually represent the same piece of physical hardware. So

just be aware that the word `hardware' in our discussion can mean something very abstract.

Compiling and loading Modules

The "hello world" example seen earlier included a brief demonstration of building a module and loading it into the system. There is, of course, a lot more to that whole process than we have seen so far. This section provides more detail on how a module author turns source code into an executing subsystem within the kernel.

Compiling Modules

The kernel is a large, standalone program with detailed and explicit requirements on how its pieces are put together. The kernel build system is a complex beast, and we just look at a tiny piece of it.

Once you have kernel set up, creating a makefile for your module is straightforward. In fact, for the "hello world" example shown earlier , a single line will suffice:

```
obj-m := hello.o
```

The assignment above states that there is one module to be built from the object file *hello.o*. The resulting module is named *hello.ko* after being built from the object file.

If, instead, you have a module called *module.ko* that is generated from two source files (called, say, *file1.c* and *file2.c*), the correct incantation would be:

```
obj-m := module.o

module-objs := file1.o file2.o
```

For a makefile like those shown above to work, it must be invoked within the context of the larger kernel build system. If your kernel source tree is located in, say, your *~/kernel-2.6* directory, the make command required to build your module (typed in the directory containing the

module source and makefile) would be:

```
make -C ~/kernel-2.6 M=`pwd` modules
```

This command starts by changing its directory to the one provided with the -C option (that is, your kernel source directory). There it finds the kernel's top-level makefile. The M= option causes that makefile to move back into your module source directory before trying to build the modules target. This target, in turn, refers to the list of modules found in the obj-m variable, which we've set to *module.o* in our examples.

Typing the previous make command can get tiresome after a while, so the kernel developers have developed a sort of makefile idiom, which makes life easier for those building modules outside of the kernel tree. The trick is to write your makefile as follows:

```
INSTALL_DIR=modules
ifneq (${KERNELRELEASE},)
    obj-m :=simple_driver.o
else
    KERNELDIR ?= /lib/modules/$(shell uname -r)/build
    PWD := $(shell pwd)
default:
    $(MAKE)  -C $(KERNELDIR) M=$(PWD) modules
    @rm -rf ${INSTALL_DIR}
    @mkdir ${INSTALL_DIR}
    @mv -f *.o *.ko *.mod.c .*.cmd ${INSTALL_DIR}
clean:
    rm -rf ${INSTALL_DIR}
endif
```

This makefile is read twice on a typical build. When the makefile is invoked from the command line, it notices that the KERNELRELEASE variable has not been set. It locates the kernel source directory by taking advantage of the fact that the symbolic link *build* in the installed modules directory points back at the kernel build tree. If you are not actually

running the kernel that you are building for, you can supply a KERNELDIR= option on the command line, set the KERNELDIR environment variable, or rewrite the line that sets KERNELDIR in the makefile. Once the kernel source tree has been found, the makefile invokes the default: target, which runs a second make command (parameterized in the makefile as $(MAKE)) to invoke the kernel build system as described previously. On the second reading, the makefile sets obj-m, and the kernel makefiles take care of actually building the module.

Loading and Unloading Modules

After the module is built, the next step is loading it into the kernel, insmod does the job for you. The program loads the module code and data into the kernel, which, in turn, performs a function similar to that of ld, in that it links any unresolved symbol in the module to the symbol table of the kernel. Unlike the linker, however, the kernel doesn't modify the module's disk file, but rather an in-memory copy. insmod accepts a number of command-line options, and it can assign values to parameters in your module before linking it to the current kernel. Thus, if a module is correctly designed, it can be configured at load time; load-time configuration gives the user more flexibility than compile-time configuration, which is still used sometimes.

insmod: it relies on a system call defined in *kernel/module.c*. The function sys_init_module allocates kernel memory to hold a module (this memory is allocated with vmalloc; it then copies the module text into that memory region, resolves kernel references in the module via the kernel symbol table, and calls the module's initialization function to get everything going.

The modprobe utility, like insmod, loads a module into the kernel. It differs in that it will look at the module to be loaded to see whether it references any symbols that are not currently defined in the kernel. If any such references are found, modprobe looks for other modules in the current module search path that define the relevant symbols. When modprobe finds those modules (which are needed by the module being loaded), it loads them into the kernel as well. If you use insmod in this

situation instead, the command fails with an "unresolved symbols" message left in the system logfile.

As mentioned before, modules may be removed from the kernel with the rmmod utility. Note that module removal fails if the kernel believes that the module is still in use, or if the kernel has been configured to disallow module removal. It is possible to configure the kernel to allow "forced" removal of modules, even when they appear to be busy.

The lsmod program produces a list of the modules currently loaded in the kernel. Some other information, such as any other modules making use of a specific module, is also provided. lsmod works by reading the */proc/modules* virtual file. Information on currently loaded modules can also be found in the sysfs virtual filesystem under */sys/module*.

The Kernel Symbol Table

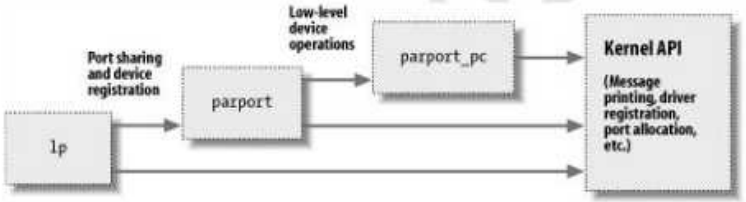
We've seen how insmod resolves undefined symbols against the table of public kernel symbols. The table contains the addresses of global kernel items—functions and variables—that are needed to implement modularized drivers. When a module is loaded, any symbol exported by the module becomes part of the kernel symbol table. In the usual case, a module implements its own functionality without the need to export any symbols at all. You need to export symbols, however, whenever other modules may benefit from using them.

New modules can use symbols exported by your module, and you can stack new modules on top of other modules. Module stacking is implemented in the mainstream kernel sources as well: the msdos filesystem relies on symbols exported by the fat module, and each input USB device module stacks on the usbcore and input modules.

Module stacking is useful in complex projects. If a new abstraction is implemented in the form of a device driver, it might offer a plug for hardware-specific implementations. For example, the video-for-linux set of

drivers is split into a generic module that exports symbols used by lower-level device drivers for specific hardware. According to your setup, you load the generic video module and the specific module for your installed hardware. Support for parallel ports and the wide variety of attachable devices is handled in the same way, as is the USB kernel subsystem. Stacking in the parallel port subsystem is shown in Figure 2-2; the arrows show the communications between the modules and with the kernel programming interface.

Figure: Stacking of parallel port driver modules



When using stacked modules, it is helpful to be aware of the modprobe utility. As we described earlier, modprobe functions in much the same way as insmod, but it also loads any other modules that are required by the module you want to load. Thus, one modprobe command can sometimes replace several invocations of insmod (although you'll still need insmod when loading your own modules from the current directory, because modprobe looks only in the standard installed module directories).

Using stacking to split modules into multiple layers can help reduce development time by simplifying each layer. This is similar to the separation between mechanism and policy that we discussed in Chapter 1.

The Linux kernel header files provide a convenient way to manage

the visibility of your symbols, thus reducing namespace pollution (filling the namespace with names that may conflict with those defined elsewhere in the kernel) and promoting proper information hiding. If your module needs to export symbols for other modules to use, the following macros should be used.

```
EXPORT_SYMBOL(name);
EXPORT_SYMBOL_GPL(name);
```

Either of the above macros makes the given symbol available outside the module. The _GPL version makes the symbol available to GPL-licensed modules only. Symbols must be exported in the global part of the module's file, outside of any function, because the macros expand to the declaration of a special-purpose variable that is expected to be accessible globally. This variable is stored in a special part of the module executible (an "ELF section") that is used by the kernel at load time to find the variables exported by the module. (Interested readers can look at *<linux/module.h>* for the details, even though the details are not needed to make things work.)

Scull Device Registration

Throughout the session, we present code fragments extracted from a real device driver: scull (Simple Character Utility for Loading Localities). scull is a char driver that acts on a memory area as though it were a device.

The advantage of scull is that it isn't hardware dependent. scull just acts on some memory, allocated from the kernel. Anyone can compile and run scull, and scull is portable across the computer architectures on which Linux runs. On the other hand, the device doesn't do anything "useful" other than demonstrate the interface between the kernel and char drivers and allow the user to run some tests.

The Design of scull

The first step of driver writing is defining the capabilities (the mechanism) the driver will offer to user programs. Since our "device" is part of the computer's memory, we're free to do what we want with it. It can be a sequential or random-access device, one device or many, and so on.

1.1: Major and Minor Numbers

Char devices are accessed through names in the filesystem. Those names are called special files or device files or simply nodes of the filesystem tree; they are conventionally located in the */dev* directory. Special files for char drivers are identified by a "c" in the first column of the output of ls -l.

EmbLogic Embedded Technologies Pvt Ltd

If you issue the ls -l command, you'll see two numbers (separated by a comma) in the device file entries before the date of the last modification, where the file length normally appears. These numbers are the major and minor device number for the particular device. The following listing shows a few devices as they appear on a typical system. Their major numbers are 1, 4, 7, and 10, while the minors are 1, 3, 5, 64, 65, and 129.

```
crw-rw-rw- 1 root     root       1,   3 Apr 11  2002 null
crw-------   1 root     root      10,   1 Apr 11  2002 psaux
crw-------   1 root     root       4,   1 Oct 28  03:04 tty1
crw-rw-rw- 1 root     tty        4,  64 Apr 11  2002 ttys0
crw-rw----   1 root     uucp       4,  65 Apr 11  2002 ttyS1
crw--w----   1 vcsa     tty        7,   1 Apr 11  2002 vcs1
crw--w----   1 vcsa     tty        7, 129 Apr 11  2002 vcsa1
crw-rw-rw- 1 root     root       1,   5 Apr 11  2002 zero
```

The major number identifies the driver associated with the device. Modern Linux kernels allow multiple drivers to share major numbers, but most devices that you will see are still organized on the one-major-one-driver principle.

The minor number is used by the kernel to determine exactly which device is being referred to. Depending on how your driver is written, you can either get a direct pointer to your device from the kernel, or you can use the minor number yourself as an index into a local array of devices. Either way, the kernel itself knows almost nothing about minor numbers beyond the fact that they refer to devices implemented by your driver.

1.2: 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);
```

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

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

Allocating and Freeing Device Numbers

One of the first things your driver will need to do when setting up a char device is to obtain one or more device numbers to work with.

The necessary function for this task is register_chrdev_region, which is declared in *<linux/fs.h>*:

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

Here, first is the beginning device number of the range you would like to allocate. Count is the total number of contiguous device numbers you are requesting. Finally, name is the name of the device that should be associated with this number range; it will appear in */proc/devices* and sysfs. As with most kernel functions, the return value from register_chrdev_region will be 0 if the allocation was successfully performed. In case of error, a negative error code will be returned, and you will not have access to the requested region.

register_chrdev_region works well if you know ahead of time exactly which device numbers you want. Often, however, you will not know which major numbers your device will use; there is a constant effort within the Linux kernel development community to move over to the use of dynamically-allocated device numbers. The kernel will happily allocate a major number for you on the fly, but you must request this allocation

by using a different function:

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

With this function, dev is an output-only parameter that will, on successful completion, hold the first number in your allocated range. firstminor should be the requested first minor number to use; it is usually 0. The count and name parameters work like those given to request_chrdev_region.

Regardless of how you allocate your device numbers, you should free them when they are no longer in use. Device numbers are freed with:

void unregister_chrdev_region(dev_t first, unsigned int count);

The usual place to call unregister_chrdev_region would be in your module's cleanup function.

The above functions allocate device numbers for your driver's use, but they do not tell the kernel anything about what you will actually do with those numbers. Before a user-space program can access one of those device numbers, your driver needs to connect them to its internal functions that implement the device's operations.

1.3: Dynamic Allocation of Major Numbers

Some major device numbers are statically assigned to the most common devices. A list of those devices can be found in *Documentation/devices.txt* within the kernel source tree. The chances of a static number having already been assigned for the use of your new driver are small, however, and new numbers are not being assigned. So, as a driver writer, you have a choice: you can simply pick a number that appears to be unused, or you can allocate major numbers in a dynamic manner. Picking a number may work as long as the only user of your

driver is you; once your driver is more widely deployed, a randomly picked major number will lead to conflicts and trouble.

Thus, for new drivers, we strongly suggest that you use dynamic allocation to obtain your major device number, rather than choosing a number randomly from the ones that are currently free. In other words, your drivers should almost certainly be using alloc_chrdev_region rather than register_chrdev_region.

The disadvantage of dynamic assignment is that you can't create the device nodes in advance, because the major number assigned to your module will vary. For normal use of the driver, this is hardly a problem, because once the number has been assigned, you can read it from */proc/devices*.

Even better device information can usually be obtained from sysfs, generally mounted on */sys* on 2.6-based systems.

To load a driver using a dynamic major number, therefore, the invocation of insmod can be replaced by a simple script that, after calling insmod, reads */proc/devices* in order to create the special file(s).

A typical */proc/devices* file looks like the following:

```
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 ttyS
  6 lp
  7 vcs
 10 misc
 13 input
 14 sound
```

```
 21 sg
180 usb

Block devices:
  2 fd
  8 sd
 11 sr
 65 sd
 66 sd
```

If you are loading and unloading only a single driver, you can just use rmmod and insmod after the first time you create the special files with your script: dynamic numbers are not randomized, and you can count on the same number being chosen each time if you don't load any other (dynamic) modules.

The best way to assign major numbers, in our opinion, is by defaulting to dynamic allocation while leaving yourself the option of specifying the major number at load time, or even at compile time. The scull implementation works in this way; it uses a global variable, scull_major, to hold the chosen number (there is also a scull_minor for the minor number). The variable is initialized to SCULL_MAJOR, defined in *scull.h*. The default value of SCULL_MAJOR in the distributed source is 0, which means "use dynamic assignment." The user can accept the default or choose a particular major number, either by modifying the macro before compiling or by specifying a value for scull_major on the insmod command line.

Here's the code we use in scull 's source to get a major number:

```
if (scull_major) {
    dev = MKDEV(scull_major, scull_minor);
    result = register_chrdev_region(dev, scull_nr_devs, "scull");
} else {
        result = alloc_chrdev_region(&dev, scull_minor, scull_nr_devs,
"scull");
```



```
    scull_major = MAJOR(dev);
}
if (result < 0) {
    printk(KERN_WARNING "scull: can't get major %d\n", scull_major);
    return result;
}
```

Almost all of the sample drivers used in this book use similar code for their major number assignment.

Character Device Files

1. Character Device Drivers

1.1 The file_operations Structure

The file_operations structure is defined in linux/fs.h, and holds pointers to functions defined by the driver that perform various operations on the device. Each field of the structure corresponds to the address of some function defined by the driver to handle a requested operation.

For example, every character driver needs to define a function that reads from the device. The file_operations structure holds the address of the module's function that performs that operation. Here is what the definition looks like for kernel 2.4.2:

```
 struct file_operations {
struct module *owner;
loff_t (*llseek) (struct file *, loff_t, int);
ssize_t (*read) (struct file *, char *, size_t, loff_t *);
ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
int (*readdir) (struct file *, void *, filldir_t);
unsigned int (*poll) (struct file *, struct poll_table_struct *);
int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned
long);
int (*mmap) (struct file *, struct vm_area_struct *);
int (*open) (struct inode *, struct file *);
int (*flush) (struct file *);
int (*release) (struct inode *, struct file *);
int (*fsync) (struct file *, struct dentry *, int datasync);
int (*fasync) (int, struct file *, int);
int (*lock) (struct file *, int, struct file_lock *);
ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, lo
ff_t *);
 ssize_t (*writev) (struct file *, const struct iovec *, unsigned long,
ff_t *);
 };
```

Some operations are not implemented by a driver. For example, a driver that handles a video card won't need to read from a directory structure. The corresponding entries in the file_operations structure should be set to NULL. There is a gcc extension that makes assigning to this structure more convenient. You'll see it in modern drivers, and may catch you by surprise. This is what the new way of assigning to the structure looks like:

```
struct file_operations fops = {
    read: device_read,
    write: device_write,
    open: device_open,
    release: device_release
};
```

A pointer to a struct file_operations is commonly named fops.

1.2. The file structure

Each device is represented in the kernel by a file structure, which is defined in linux/fs.h. Be aware that a file is a kernel level structure and never appears in a user space program. It's not the same thing as a FILE, which is defined by glibc and would never appear in a kernel space function. Also, its name is a bit misleading; it represents an abstract open `file', not a file on a disk, which is represented by a structure named inode.

A pointer to a struct file is commonly named filp. You'll also see it refered to as struct file file. Go ahead and look at the definition of file. Most of the entries you see, like struct dentry aren't used by device drivers, and you can ignore them. This is because drivers don't fill file

directly; they only use structures contained in file which are created elsewhere.

1.3. Registering A Device

As discussed earlier, char devices are accessed through device files, usually located in /dev. The major number tells you which driver handles which device file. The minor number is used only by the driver itself to differentiate which device it's operating on, just in case the driver handles more than one device. Adding a driver to your system means registering it with the kernel. This is synonymous with assigning it a major number during the module's initialization. You do this by using the register_chrdev function, defined by linux/fs.h.

```
int register_chrdev(unsigned int major,
                    const char *name,
                    struct file_operations *fops);
```

where unsigned int major is the major number you want to request, const char *name is the name of the device as it'll appear in /proc/devices and struct file_operations *fops is a pointer to the file_operations table for your driver. A negative return value means the registertration failed. Note that we didn't pass the minor number to register_chrdev. That's because the kernel doesn't care about the minor number; only our driver uses it. You can ask the kernel to assign you a dynamic major number. If you pass a major number of 0 to register_chrdev, the return value will be the dynamically allocated major number. The downside is that you can't make a device file in advance, since you don't know what the major number will be. There are a couple of ways to do this. First, the driver itself can print the newly assigned number and we can make the device file by hand. Second, the newly registered device will have an entry in /proc/devices, and we can either make the device file by hand or write a shell script to read the file in and make the device file. The third

method is we can have our driver make the the device file using the mknod system call after a successful registration and rm during the call to cleanup_module.

1.4. Unregistering A Device

We can't allow the kernel module to be rmmod'ed whenever root feels like it. If the device file is opened by a process and then we remove the kernel module, using the file would cause a call to the memory location where the appropriate function (read/write) used to be. If we're lucky, no other code was loaded there, and we'll get an ugly error message. If we're unlucky, another kernel module was loaded into the same location, which means a jump into the middle of another function within the kernel. The results of this would be impossible to predict, but they can't be very positive. Normally, when you don't want to allow something, you return an error code (a negative number) from the function which is supposed to do it. With cleanup_module that's impossible because it's a void function.

However, there's a counter which keeps track of how many processes are using your module. You can see what it's value is by looking at the 3rd field of /proc/modules. If this number isn't zero, rmmod will fail. Note that you don't have to check the counter from within cleanup_module because the check will be performed for you by the system call sys_delete_module, defined in linux/module.c. You shouldn't use this counter directly, but there are macros defined in linux/modules.h which let you increase, decrease and display this counter:

- MOD_INC_USE_COUNT: Increment the use count.
- MOD_DEC_USE_COUNT: Decrement the use count.
- MOD_IN_USE: Display the use count.

It's important to keep the counter accurate; if you ever do lose track of the correct usage count, you'll never be able to unload the module; it's now reboot time, boys and girls. This is bound to happen to you sooner or later during a module's development.

1.5. chardev.c

The next code sample creates a char driver named chardev. You can cat its device file (or open the file with a program) and the driver will put the number of times the device file has been read from into the file. We don't support writing to the file (like echo "hi" > /dev/hello), but catch these attempts and tell the user that the operation isn't supported. Don't worry if you don't see what we do with the data we read into the buffer; we don't do much with it. We simply read in the data and print a message acknowledging that we received it.

Example 4-1. chardev.c

```
/*chardev.c: Creates a read-only char device that says how many times
you've read from the dev file  */

    #if defined(CONFIG_MODVERSIONS) && ! defined(MODVERSIONS)
    #include <linux/modversions.h>
    #define MODVERSIONS
    #endif
    #include <linux/kernel.h>
    #include <linux/module.h>
    #include <linux/fs.h>
    #include <asm/uaccess.h>                    /* for put_user */
            /* Prototypes - this would normally go in a .h file   */
    int init_module(void);
    void cleanup_module(void);
    static int device_open(struct inode *, struct file *);
    static int device_release(struct inode *, struct file *);
    static ssize_t device_read(struct file *, char *, size_t, loff_t *);
    static ssize_t device_write(struct file *, const char *, size_t, loff_t*);
```

```
#define SUCCESS 0
#define DEVICE_NAME "chardev"
                        /* Dev name as it appears in /proc/devices*/
#define BUF_LEN   /* Max length of the message from the device */
/* Global variables are declared as static, so are global within the file. */
static int Major;   /* Major number assigned to our device driver  */
static int Device_Open = 0;
/* Is device open? Used to prevent multiple access to the device   */
static char msg[BUF_LEN];
                    /* The msg the device will give when asked     */
static char *msg_Ptr;
static struct file_operations fops = {
.read = device_read,
.write = device_write,
.open = device_open,
.release = device_release
};
  /*                Functions                */
int init_module(void)
{
    Major = register_chrdev(0, DEVICE_NAME, &fops);
    if (Major < 0) {
        printk ("Registering the character device failed with %d\n",
                Major);
        return Major;
    }
    printk("<1>I was assigned major number %d. To talk to the
            driver, \n", Major);
    printk("<1>create a dev file with mknod /dev/hello c %d 0'.\n",
            Major);
    printk("<1>Try various minor numbers. Try to cat and echo to
            the device file.\n");
    printk("<1>Remove the device file and module when done.\n");
    return 0;
```



```
}
void cleanup_module(void)
{
    /* Unregister the device */
    int ret = unregister_chrdev(Major, DEVICE_NAME);
    if (ret < 0) printk("Error in unregister_chrdev: %d\n", ret);
}
/*                Methods                */
/* Called when a process tries to open the device file, like
     "cat /dev/mycharfile"                */
static int device_open(struct inode *inode, struct file *file)
{
      static int counter = 0;
      if (Device_Open) return -EBUSY;
      Device_Open++;
      sprintf(msg,"I already told you %d times Hello world!\n",
              counter++");
      msg_Ptr = msg;
      MOD_INC_USE_COUNT;
      return SUCCESS;
}
/* Called when a process closes the device file.          */
static int device_release(struct inode *inode, struct file *file)
{
      Device_Open --; /* We're now ready for our next caller */
      /* Decrement the usage count, or else once you opened the
      file, you'll never get get rid of the module. */
      MOD_DEC_USE_COUNT;
      return 0;
}
/* Called when a process, which already opened the dev file,
attempts to read from it.          */
static ssize_t device_read(struct file *filp, char *buffer,  size_t
                           length,  loff_t *offset)
```

```
{
    /* Number of bytes actually written to the buffer */
    int bytes_read = 0;
    /* If we're at the end of the message, return 0 signifying
    end of file */
    if (*msg_Ptr == 0) return 0;
    /* Actually put the data into the buffer */
    while (length && *msg_Ptr) {
    /* The buffer is in the user data segment, not the kernel
    segment; assignment won't work. We have to use put_user
    which copies data from the kernel data segment to the user
    data segment. */
    put_user(*(msg_Ptr++), buffer++);
    length--;
    bytes_read++;
    }
    /* Most read functions return the number of bytes put into
        the buffer */
    return bytes_read;
    }
    /* Called when a process writes to dev file: echo "hi" >
        /dev/hello */
    static ssize_t device_write(struct file *filp, const char *buff,
            size_t len, loff_t *off)
{
    printk ("<1>Sorry, this operation isn't supported.\n");
    return -EINVAL;
}
```

Device Drivers Preliminaries

1.1: Module Parameters

Several parameters that a driver needs to know can change from system to system. These can vary from the device number to use to numerous aspects of how the driver should operate. For example, drivers for SCSI adapters often have options controlling the use of tagged command queuing, and the Integrated Device Electronics (IDE) drivers allow user control of DMA operations. If your driver controls older hardware, it may also need to be told explicitly where to find that hardware's I/O ports or I/O memory addresses. The kernel supports these needs by making it possible for a driver to designate parameters that may be changed when the driver's module is loaded.

These parameter values can be assigned at load time by insmod or modprobe ; the latter can also read parameter assignment from its configuration file (*/etc/modprobe.conf*). The commands accept the specification of several types of values on the command line:

```
insmod hellop howmany=10 whom="Mom"
```

Upon being loaded that way, hellop would say "Hello, Mom" 10 times.

However, before insmod can change module parameters, the module must make them available. Parameters are declared with the macro, which is defined in *moduleparam.h*. `module_param` takes three parameters: the name of the variable, its type, and a permissions mask to be used for an accompanying sysfs entry. The macro should be placed outside of any function and is typically found near the head of the source file.

```
static char *whom = "world";
static int howmany = 1;
module_param(howmany, int, S_IRUGO);
module_param(whom, charp, S_IRUGO);
```

Numerous types are supported for module parameters:

bool
invbool

A boolean (true or false) value (the associated variable should be of type `int`). The `invbool` type inverts the value, so that true values become false and vice versa.

charp

A char pointer value. Memory is allocated for user-provided strings, and the pointer is set accordingly.

int
long
short
uint
ulong
ushort

Basic integer values of various lengths. The versions starting with `u` are for unsigned values.

Array parameters, where the values are supplied as a comma-separated list, are also supported by the module loader. To declare an array parameter, use:

```
module_param_array(name,type,num,perm);
```

Where `name` is the name of your array (and of the parameter), `type` is the type of the array elements, `num` is an integer variable, and `perm` is the usual permissions value. If the array parameter is set at load time, `num` is set to the number of values supplied. The module loader refuses to accept more values than will fit in the array.

All module parameters should be given a default value; insmod changes the value only if explicitly told to by the user. The module can check for explicit parameters by testing parameters against their default values.

The final module_param field is a permission value; you should use the definitions found in *<linux/stat.h>*. This value controls who can access the representation of the module parameter in sysfs. If `perm` is set to `0`, there is no sysfs entry at all; otherwise, it appears under */sys/module* with the given set of permissions. Use `S_IRUGO` for a parameter that can be read by the world but cannot be changed; `S_IRUGO|S_IWUSR` allows root to change the parameter. Note that if a parameter is changed by sysfs, the value of that parameter as seen by your module changes, but your module is not notified in any other way. You should probably not make module parameters writable, unless you are prepared to detect the change and react accordingly.

EmbLogic Embedded Technologies Pvt Ltd

Building Running Modules

spatlou@googlegroups.com

This session explains all the essential concepts about modules and kernel programming. In these few pages, we build and run a complete module, and look at some of the basic code shared by all modules.

Setting Up Your Test System

We recommend that you obtain a "mainline" kernel from the kernel.org mirror network, and install it on your system. Vendor kernels can be heavily patched and divergent from the mainline.

Regardless of the origin of your kernel, building modules for 2.6.x requires that you have a configured and built kernel tree on your system. So you shoould first come up with a kernel source tree, build a new kernel, and install it on your system.

The Hello World Module

The following code is a complete "hello world" module:

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void)
{
    printk(KERN_ALERT "Hello, world\n");
    return 0;
}
static void hello_exit(void)
{
    printk(KERN_ALERT "Goodbye, cruel world\n");
```

```
}
module_init(hello_init);
module_exit(hello_exit);
```

This module defines two functions, one to be invoked when the module is loaded into the kernel (hello_init) and one for when the module is removed (hello_exit). The module_init and module_exit lines use special kernel macros to indicate the role of these two functions. Another special macro (MODULE_LICENSE) is used to tell the kernel that this module bears a free license; without such a declaration, the kernel complains when the module is loaded.

The printk function is defined in the Linux kernel and made available to modules. The kernel needs its own printing function because it runs by itself, without the help of the C library. The module can call printk because, after insmod has loaded it, the module is linked to the kernel and can access the kernel's public symbols. The string `KERN_ALERT` is the priority of the message. We've specified a high priority in this module, because a message with the default priority might not show up anywhere useful.

The priority is just a string, such as `<1>`, which is prepended to the printk format string. Note the lack of a comma after `KERN_ALERT`; adding a comma there is a common and annoying typo.

Run the Makefile script given below to compile the source.

```
STALL_DIR=modules
ifneq (${KERNELRELEASE},)
    obj-m :=simple_driver.o

else
```



```
    KERNELDIR ?= /lib/modules/$(shell uname -r)/build
    PWD := $(shell pwd)
default:
    $(MAKE)  -C $(KERNELDIR) M=$(PWD) modules
    @rm -rf ${INSTALL_DIR}
    @mkdir ${INSTALL_DIR}
    @mv -f *.o *.ko *.mod.c .*.cmd ${INSTALL_DIR}
clean:
    rm -rf ${INSTALL_DIR}
endif
```

You can test the module with the insmod and rmmod utilities, as shown below. Note that only the superuser can load and unload a module.

```
% su -
password:
root# make
root# insmod ./modules/hello.ko
root# dmesg
Hello, world
root# rmmod hello
root# dmesg
Goodbye cruel world
root#
```

Please note once again that, for the above sequence of commands to work, you must have a properly configured and built kernel tree in a place where the makefile is able to find it (*/usr/src/linux-2.6.10* in the example shown).

The prrintk message goes to one of the system log files, such as */var/log/messages, that is displayed using dmesg at the shell.*

Kernel Modules Versus Applications

While most small and medium-sized applications perform a single task from beginning to end, every kernel module just registers itself in order to serve future requests, and its initialization function terminates immediately. In other words, the task of the module's initialization function is to prepare for later invocation of the module's functions. The module's exit function (hello_exit in the example) gets invoked just before the module is unloaded. This kind of approach to programming is similar to event-driven programming, but while not all applications are event-driven, each and every kernel module is.

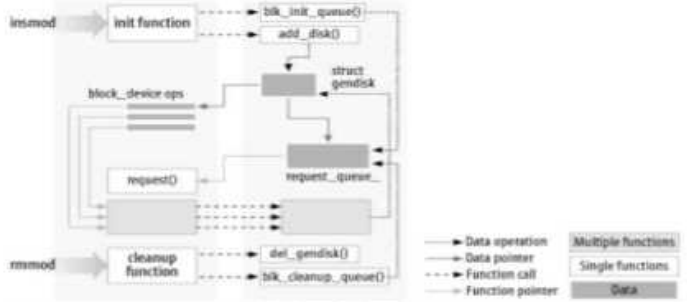
Another major difference between event-driven applications and kernel code is in the exit function: whereas an application that terminates can be lazy in releasing resources or avoids clean up altogether, the exit function of a module must carefully undo everything the init function built up, or the pieces remain around until the system is rebooted.

The ability to unload a module is one of the features of modularization that you'll most appreciate, because it helps cut down development time; you can test successive versions of your new driver without going through the lengthy shutdown/reboot cycle each time.

As a programmer, you know that an application can call functions it doesn't define: the linking stage resolves external references using the appropriate library of functions. printf is one of those callable functions and is defined in libc. A module, on the other hand, is linked only to the kernel, and the only functions it can call are the ones exported by the kernel; there are no libraries to link to. The printk function used in *hello.c* earlier, for example, is the version of printf defined within the kernel and exported to modules. It behaves similarly to the original function, with a few minor differences, the main one being lack of floating-point support.

Figure 1 shows how function calls and function pointers are used in a module to add new functionality to a running kernel.

Figure 1. Linking a module to the kernel



Only functions that are actually part of the kernel itself may be used in kernel modules. Anything related to the kernel is declared in headers found in the kernel source tree you have set up and configured; most of the relevant headers live in *include/linux* and *include/asm*, but other subdirectories of *include* have been added to host material associated to specific kernel subsystems.

The role of individual kernel headers is introduced as and when needed.

Another important difference between kernel programming and application programming is in how each environment handles faults: whereas a segmentation fault is harmless during application development and a debugger can always be used to trace the error to the problem in the source code, a kernel fault kills the current process at least, if not the whole system. We see how to trace kernel errors later.

User Space and Kernel Space

A module runs in kernel space, whereas applications run in user space. This concept is at the base of operating systems theory.

The role of the operating system, in practice, is to provide programs with a consistent view of the computer's hardware. In addition, the operating system must account for independent operation of programs and protection against unauthorized access to resources. This nontrivial task is possible only if the CPU enforces protection of system software from the applications.

Under Unix, the kernel executes in the highest level (also called supervisor mode), where everything is allowed, whereas applications execute in the lowest level (the so-called user mode), where the processor regulates direct access to hardware and unauthorized access to memory.

We usually refer to the execution modes as kernel space and user space. These terms encompass not only the different privilege levels inherent in the two modes, but also the fact that each mode can have its own memory mapping—its own address space—as well.

Unix transfers execution from user space to kernel space whenever an application issues a system call or is suspended by a hardware interrupt. Kernel code executing a system call is working in the context of a process—it operates on behalf of the calling process and is able to access data in the process's address space. Code that handles interrupts, on the other hand, is asynchronous with respect to processes and is not related to any particular process.

The role of a module is to extend kernel functionality; modularized code runs in kernel space. Usually a driver performs both the tasks

outlined previously: some functions in the module are executed as part of system calls, and some are in charge of interrupt handling.

Concurrency in the Kernel

One way in which kernel programming differs greatly from conventional application programming is the issue of concurrency. Most applications, with the notable exception of multithreading applications, typically run sequentially, from the beginning to the end, without any need to worry about what else might be happening to change their environment. Kernel code does not run in such a simple world, and even the simplest kernel modules must be written with the idea that many things can be happening at once.

There are a few sources of concurrency in kernel programming. Naturally, Linux systems run multiple processes, more than one of which can be trying to use your driver at the same time. Most devices are capable of interrupting the processor; interrupt handlers run asynchronously and can be invoked at the same time that your driver is trying to do something else. Several software abstractions (such as kernel timers) run asynchronously as well. Finally, in 2.6, kernel code has been made preemptible; this change causes even uniprocessor systems to have many of the same concurrency issues as multiprocessor systems.

As a result, Linux kernel code, including driver code, must be *reentrant*—it must be capable of running in more than one context at the same time. Data structures must be carefully designed to keep multiple threads of execution separate, and the code must take care to access shared data in ways that prevent corruption of the data. Writing code that handles concurrency and avoids race conditions requires thought and can be tricky. Proper management of concurrency is required to write correct kernel code; for that reason, every sample driver we try has been written with concurrency in mind.

A common mistake made by driver programmers is to assume that concurrency is not a problem as long as a particular segment of code does not go to sleep (or "block"). In 2.6, kernel code can (almost) never assume that it can hold the processor over a given stretch of code. If you do not write your code with concurrency in mind, it will be subject to catastrophic failures that can be exceedingly difficult to debug.

The Current Process

Although kernel modules don't execute sequentially as applications do, most actions performed by the kernel are done on behalf of a specific process. Kernel code can refer to the current process by accessing the global item `current`, defined in *<asm/current.h>*, which yields a pointer to `struct task_struct`, defined by *<linux/sched.h>*. The `current` pointer refers to the process that is currently executing. During the execution of a system call, such as open or read, the current process is the one that invoked the call. Kernel code can use process-specific information by using `current`, if it needs to do so.

The following statement prints the process ID and the command name of the current process by accessing certain fields in `struct task_struct`:

```
printk(KERN_INFO "The process is \"%s\" (pid %i)\n",current->comm, current->pid);
```

The command name stored in `current->comm` is the base name of the program file that is being executed by the current process.

Few Other Details

Kernel programming differs from user-space programming in many

Building and Running Modules

ways. So, as you dig into the kernel, the following issues should be kept in mind.

Applications are laid out in virtual memory with a very large stack area. The stack, of course, is used to hold the function call history and all automatic variables created by currently active functions. The kernel, instead, has a very small stack; it can be as small as a single, 4096-byte page. Your functions must share that stack with the entire kernel-space call chain. Thus, it is never a good idea to declare large automatic variables; if you need larger structures, you should allocate them dynamically at call time.

Often, as you look at the kernel API, you will encounter function names starting with a double underscore (__ _). Functions so marked are generally a low-level component of the interface.

Kernel code cannot do floating point arithmetic. Enabling floating point would require that the kernel save and restore the floating point processor's state on each entry to, and exit from, kernel space—at least, on some architectures. Given that there really is no need for floating point in kernel code, the extra overhead is not worthwhile.

EmbLogic Embedded Technologies Pvt Ltd

Device Drivers Preliminaries

1.1: Module Parameters

Several parameters that a driver needs to know can change from system to system. These can vary from the device number to use to numerous aspects of how the driver should operate. For example, drivers for SCSI adapters often have options controlling the use of tagged command queuing, and the Integrated Device Electronics (IDE) drivers allow user control of DMA operations. If your driver controls older hardware, it may also need to be told explicitly where to find that hardware's I/O ports or I/O memory addresses. The kernel supports these needs by making it possible for a driver to designate parameters that may be changed when the driver's module is loaded.

These parameter values can be assigned at load time by insmod or modprobe ; the latter can also read parameter assignment from its configuration file (*/etc/modprobe.conf*). The commands accept the specification of several types of values on the command line:

```
insmod hellop howmany=10 whom="Mom"
```

Upon being loaded that way, hellop would say "Hello, Mom" 10 times.

However, before insmod can change module parameters, the module must make them available. Parameters are declared with the macro, which is defined in *moduleparam.h*. `module_param` takes three parameters: the name of the variable, its type, and a permissions mask to be used for an accompanying sysfs entry. The macro should be placed outside of any function and is typically found near the head of the source file.

```
static char *whom = "world";
static int howmany = 1;
module_param(howmany, int, S_IRUGO);
module_param(whom, charp, S_IRUGO);
```

Numerous types are supported for module parameters:

bool
invbool

A boolean (true or false) value (the associated variable should be of type `int`). The `invbool` type inverts the value, so that true values become false and vice versa.

charp

A char pointer value. Memory is allocated for user-provided strings, and the pointer is set accordingly.

int
long
short
uint
ulong
ushort

Basic integer values of various lengths. The versions starting with `u` are for unsigned values.

Array parameters, where the values are supplied as a comma-separated list, are also supported by the module loader. To declare an array parameter, use:

```
module_param_array(name,type,num,perm);
```

Where `name` is the name of your array (and of the parameter), `type` is the type of the array elements, `num` is an integer variable, and `perm` is the usual permissions value. If the array parameter is set at load time, `num` is set to the number of values supplied. The module loader refuses to accept more values than will fit in the array.

All module parameters should be given a default value; insmod changes the value only if explicitly told to by the user. The module can check for explicit parameters by testing parameters against their default values.

The final module_param field is a permission value; you should use the definitions found in *<linux/stat.h>*. This value controls who can access the representation of the module parameter in sysfs. If `perm` is set to `0`, there is no sysfs entry at all; otherwise, it appears under */sys/module* with the given set of permissions. Use `S_IRUGO` for a parameter that can be read by the world but cannot be changed; `S_IRUGO|S_IWUSR` allows root to change the parameter. Note that if a parameter is changed by sysfs, the value of that parameter as seen by your module changes, but your module is not notified in any other way. You should probably not make module parameters writable, unless you are prepared to detect the change and react accordingly.

EmbLogic Embedded Technologies Pvt Ltd

Scull Device Registration

Throughout the session, we present code fragments extracted from a real device driver: scull (Simple Character Utility for Loading Localities). scull is a char driver that acts on a memory area as though it were a device.

The advantage of scull is that it isn't hardware dependent. scull just acts on some memory, allocated from the kernel. Anyone can compile and run scull, and scull is portable across the computer architectures on which Linux runs. On the other hand, the device doesn't do anything "useful" other than demonstrate the interface between the kernel and char drivers and allow the user to run some tests.

The Design of scull

The first step of driver writing is defining the capabilities (the mechanism) the driver will offer to user programs. Since our "device" is part of the computer's memory, we're free to do what we want with it. It can be a sequential or random-access device, one device or many, and so on.

1.1: Major and Minor Numbers

Char devices are accessed through names in the filesystem. Those names are called special files or device files or simply nodes of the filesystem tree; they are conventionally located in the */dev* directory. Special files for char drivers are identified by a "c" in the first column of the output of ls -l.

If you issue the ls -l command, you'll see two numbers (separated by a comma) in the device file entries before the date of the last modification, where the file length normally appears. These numbers are the major and minor device number for the particular device. The following listing shows a few devices as they appear on a typical system. Their major numbers are 1, 4, 7, and 10, while the minors are 1, 3, 5, 64, 65, and 129.

```
crw-rw-rw- 1 root     root       1,   3 Apr 11  2002 null
crw-------   1 root     root      10,   1 Apr 11  2002 psaux
crw-------   1 root     root       4,   1 Oct 28  03:04 tty1
crw-rw-rw- 1 root     tty        4,  64 Apr 11  2002 ttys0
crw-rw----   1 root     uucp       4,  65 Apr 11  2002 ttyS1
crw--w----   1 vcsa     tty        7,   1 Apr 11  2002 vcs1
crw--w----   1 vcsa     tty        7, 129 Apr 11  2002 vcsa1
crw-rw-rw- 1 root     root       1,   5 Apr 11  2002 zero
```

The major number identifies the driver associated with the device. Modern Linux kernels allow multiple drivers to share major numbers, but most devices that you will see are still organized on the one-major-one-driver principle.

The minor number is used by the kernel to determine exactly which device is being referred to. Depending on how your driver is written, you can either get a direct pointer to your device from the kernel, or you can use the minor number yourself as an index into a local array of devices. Either way, the kernel itself knows almost nothing about minor numbers beyond the fact that they refer to devices implemented by your driver.

1.2: 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);
```

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

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

Allocating and Freeing Device Numbers

One of the first things your driver will need to do when setting up a char device is to obtain one or more device numbers to work with.

The necessary function for this task is register_chrdev_region, which is declared in *<linux/fs.h>*:

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

Here, first is the beginning device number of the range you would like to allocate. Count is the total number of contiguous device numbers you are requesting. Finally, name is the name of the device that should be associated with this number range; it will appear in */proc/devices* and sysfs. As with most kernel functions, the return value from register_chrdev_region will be 0 if the allocation was successfully performed. In case of error, a negative error code will be returned, and you will not have access to the requested region.

register_chrdev_region works well if you know ahead of time exactly which device numbers you want. Often, however, you will not know which major numbers your device will use; there is a constant effort within the Linux kernel development community to move over to the use of dynamically-allocated device numbers. The kernel will happily allocate a major number for you on the fly, but you must request this allocation

by using a different function:

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

With this function, dev is an output-only parameter that will, on successful completion, hold the first number in your allocated range. firstminor should be the requested first minor number to use; it is usually 0. The count and name parameters work like those given to request_chrdev_region.

Regardless of how you allocate your device numbers, you should free them when they are no longer in use. Device numbers are freed with:

```
void unregister_chrdev_region(dev_t first, unsigned int count);
```

The usual place to call unregister_chrdev_region would be in your module's cleanup function.

The above functions allocate device numbers for your driver's use, but they do not tell the kernel anything about what you will actually do with those numbers. Before a user-space program can access one of those device numbers, your driver needs to connect them to its internal functions that implement the device's operations.

1.3: Dynamic Allocation of Major Numbers

Some major device numbers are statically assigned to the most common devices. A list of those devices can be found in *Documentation/devices.txt* within the kernel source tree. The chances of a static number having already been assigned for the use of your new driver are small, however, and new numbers are not being assigned. So, as a driver writer, you have a choice: you can simply pick a number that appears to be unused, or you can allocate major numbers in a dynamic manner. Picking a number may work as long as the only user of your

driver is you; once your driver is more widely deployed, a randomly picked major number will lead to conflicts and trouble.

Thus, for new drivers, we strongly suggest that you use dynamic allocation to obtain your major device number, rather than choosing a number randomly from the ones that are currently free. In other words, your drivers should almost certainly be using alloc_chrdev_region rather than register_chrdev_region.

The disadvantage of dynamic assignment is that you can't create the device nodes in advance, because the major number assigned to your module will vary. For normal use of the driver, this is hardly a problem, because once the number has been assigned, you can read it from */proc/devices*.

Even better device information can usually be obtained from sysfs, generally mounted on */sys* on 2.6-based systems.

To load a driver using a dynamic major number, therefore, the invocation of insmod can be replaced by a simple script that, after calling insmod, reads */proc/devices* in order to create the special file(s).

A typical */proc/devices* file looks like the following:

```
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 ttyS
  6 lp
  7 vcs
 10 misc
 13 input
 14 sound
```

```
 21 sg
180 usb

Block devices:
  2 fd
  8 sd
 11 sr
 65 sd
 66 sd
```

If you are loading and unloading only a single driver, you can just use rmmod and insmod after the first time you create the special files with your script: dynamic numbers are not randomized, and you can count on the same number being chosen each time if you don't load any other (dynamic) modules.

The best way to assign major numbers, in our opinion, is by defaulting to dynamic allocation while leaving yourself the option of specifying the major number at load time, or even at compile time. The scull implementation works in this way; it uses a global variable, scull_major, to hold the chosen number (there is also a scull_minor for the minor number). The variable is initialized to SCULL_MAJOR, defined in *scull.h*. The default value of SCULL_MAJOR in the distributed source is 0, which means "use dynamic assignment." The user can accept the default or choose a particular major number, either by modifying the macro before compiling or by specifying a value for scull_major on the insmod command line.

Here's the code we use in scull 's source to get a major number:

```
if (scull_major) {
    dev = MKDEV(scull_major, scull_minor);
    result = register_chrdev_region(dev, scull_nr_devs, "scull");
} else {
        result = alloc_chrdev_region(&dev, scull_minor, scull_nr_devs,
"scull");
```



```
    scull_major = MAJOR(dev);
}
if (result < 0) {
    printk(KERN_WARNING "scull: can't get major %d\n", scull_major);
    return result;
}
```

Almost all of the sample drivers used in this book use similar code for their major number assignment.

Data Structures

1.1: Some Important Data Structures

Device number registration is just the first of many tasks that driver code must carry out. Most of the fundamental driver operations involve three important kernel data structures, called file_operations, file, and inode.

1.2: File Operations

The file_operations structure is used by the char driver, to sets up connections between driver operations and the major number. The structure, defined in *<linux/fs.h>*, is a collection of function pointers. Each open file is associated with its own set of functions. The operations are mostly in charge of implementing the system calls and are therefore, named open, read, and so on. We can consider the file to be an "object" and the functions operating on it to be its "methods," using object-oriented programming terminology to denote actions declared by an object to act on itself.

Conventionally, a file_operations structure or a pointer to one is called fops. Each field in the structure must point to the function in the driver that implements a specific operation, or be left NULL for unsupported 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.

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 in potentially unpredictable ways.

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.

Kernel Data Structures

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 non-blocking 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.

int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);

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 (*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.[5]

[5] 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.

Kernel Data Structures

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.

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.

The scull device driver implements only the most important device methods. Its file_operations structure is initialized as follows:

```
struct file_operations scull_fops = {
    .owner =    THIS_MODULE,
    .llseek =   scull_llseek,
    .read =     scull_read,
    .write =    scull_write,
    .ioctl =    scull_ioctl,
```

Kernel Data Structures

```
    .open =     scull_open,
    .release =  scull_release,
};
```

This declaration uses the standard C tagged structure initialization syntax.

1.3: The file Structure

Struct file, defined in *<linux/fs.h>*, is the second most important data structure used in device drivers. Note that a file has nothing to do with the FILE pointers of user-space programs. A FILE is defined in the C library and never appears in kernel code. A struct file, on the other hand, is a kernel structure that never appears in user programs.

The file structure represents an open file . (It is not specific to device drivers; every open file in the system has an associated struct file in kernel space.) It is created by the kernel on open and is passed to any function that operates on the file, until the last close. After all instances of the file are closed, the kernel releases the data structure.

In the kernel sources, a pointer to struct file is usually called either file or filp ("file pointer"). We'll consistently call the pointer filp to prevent ambiguities with the structure itself. Thus, file refers to the structure and filp to a pointer to the structure.

The most important fields of struct file are shown here.

mode_t f_mode;

The file mode identifies the file as either readable or writable, by means of the bits FMODE_READ and FMODE_WRITE. You might want to check this field for read/write permission in your open or ioctl function, but you don't need to check permissions for read and write,

because the kernel checks before invoking your method. An attempt to read or write when the file has not been opened for that type of access is rejected without the driver even knowing about it.

loff_t f_pos;

The current reading or writing position. loff_t is a 64-bit value on all platforms (long long in gcc terminology). The driver can read this value if it needs to know the current position in the file but should not normally change it; read and write should update a position using the pointer they receive as the last argument instead of acting on filp->f_pos directly. The one exception to this rule is in the llseek method, the purpose of which is to change the file position.

unsigned int f_flags;

These are the file flags, such as O_RDONLY, O_NONBLOCK, and O_SYNC. A driver should check the O_NONBLOCK flag to see if nonblocking operation has been requested, the other flags are seldom used. In particular, read/write permission should be checked using f_mode rather than f_flags. All the flags are defined in the header *<linux/fcntl.h>*.

struct file_operations *f_op;

The operations associated with the file. The kernel assigns the pointer as part of its implementation of open and then reads it when it needs to dispatch any operations. The value in filp->f_op is never saved by the kernel for later reference; this means that you can change the file operations associated with your file, and the new methods will be effective after you return to the caller. For example, the code for open associated with major number 1 (*/dev/null*, */dev/zero*, and so on) substitutes the operations in filp->f_op depending on the minor number being opened. This practice allows the implementation of several behaviors under the same major number without introducing overhead at each system call. The ability to replace the file

Kernel Data Structures

operations is the kernel equivalent of "method overriding" in object-oriented programming.

void *private_data;

The open system call sets this pointer to NULL before calling the open method for the driver. You are free to make its own use of the field or to ignore it; you can use the field to point to allocated data, but then you must remember to free that memory in the release method before the file structure is destroyed by the kernel. private_data is a useful resource for preserving state information across system calls and is used by most of our sample modules.

struct dentry *f_dentry;

The directory entry (*dentry*) structure associated with the file. Device driver writers normally need not concern themselves with dentry structures, other than to access the inode structure as filp->f_dentry->d_inode.

The real structure has a few more fields, but they aren't useful to device drivers. We can safely ignore those fields, because drivers never create file structures; they only access structures created elsewhere.

1.4: The inode Structure

The *inode* structure is used by the kernel internally to represent files. Therefore, it is different from the file structure that represents an open file descriptor. There can be numerous file structures representing multiple open descriptors on a single file, but they all point to a single inode structure.

The inode structure contains a great deal of information about the file. As a general rule, only two fields of this structure are of interest for writing driver code:

dev_t i_rdev;

For inodes that represent device files, this field contains the actual device number.

struct cdev *i_cdev;

struct cdev is the kernel's internal structure that represents char devices; this field contains a pointer to that structure when the inode refers to a char device file.

There are two macros that can be used to obtain the major and minor number from an inode:

unsigned int iminor(struct inode *inode);

unsigned int imajor(struct inode *inode);

Character Device Registration

The kernel uses structures of type struct cdev to represent char devices internally. Before the kernel invokes your device's operations, you must allocate and register one or more of these structures. To do so, your code should include *<linux/cdev.h>*, where the structure and its associated helper functions are defined.

```
struct cdev {
        struct kobject kobj;
        struct module *owner;
        const struct file_operations *ops;
        struct list_head list;
        dev_t dev;
        unsigned int count;
};
```

First task is to initialize this structure.

```
struct cdev *my_cdev = cdev_alloc( );

my_cdev->ops = &my_fops;
```

The cdev structure should be embedded within a device-specific structure of your own; that is what scull does. In that case, you should initialize the structure that you have already allocated with:

```
void cdev_init(struct cdev *cdev, struct file_operations *fops);
```

Either way, there is one other struct cdev field that you need to initialize. Like the file_operations structure, struct cdev has an owner field that should be set to THIS_MODULE.

Once the cdev structure is set up, the final step is to tell the kernel about it with a call to:

```
int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
```

dev is the cdev structure, num is the first device number to which this device responds, and count is the number of device numbers that should be associated with the device.

The cdev_add call can fail. If it returns a negative error code, your device has not been added to the system. As soon as cdev_add succeeds and returns, your device is "live" and its operations can be called by the kernel. You should not call cdev_add until your driver is completely ready to handle operations on the device.

To remove a char device from the system, call:

```
void cdev_del(struct cdev *dev);
```

Clearly, you should not access the cdev structure after passing it to cdev_del.

3.4.1. Device Registration in scull

Internally, scull represents each device with a structure of type struct scull_dev. This structure is defined as:

```
struct scull_dev {
   struct scull_qset *data;        /* Pointer to first quantum set */
   int quantum;                    /* the current quantum size */
   int qset;                       /* the current array size */
   unsigned long size;             /* amount of data stored here */
   unsigned int access_key;        /* used by sculluid and scullpriv */
   struct semaphore sem;           /* mutual exclusion semaphore     */
   struct cdev cdev;               /* Char device structure      */
};
```

The struct cdev that interfaces our device to the kernel. This structure must be initialized and added to the system as described above; the scull code that handles this task is:

```
static void scull_setup_cdev(struct scull_dev *dev, int index)
```



```
{
   int err, devno = MKDEV(scull_major, scull_minor + index);
   cdev_init(&dev->cdev, &scull_fops);
   dev->cdev.owner = THIS_MODULE;
   dev->cdev.ops = &scull_fops;
   err = cdev_add (&dev->cdev, devno, 1);
   /* Fail gracefully if need be */
   if (err)
   printk(KERN_NOTICE "Error %d adding scull%d", err, index);
}
```

Since the cdev structure is embedded within struct scull_dev, cdev_init must be called to perform the initialization of that structure.

Debugging Techniques

Kernel programming brings its own, unique debugging challenges. Kernel code cannot be easily executed under a debugger, nor can it be easily traced, because it is a set of functionalities not related to any specific process. Kernel code errors can also be exceedingly hard to reproduce and can bring down the entire system with them, thus destroying much of the evidence that could be used to track them down.

printk

The most common debugging technique is monitoring using printk. printk classifies messages according to their severity by associating different loglevels, or priorities, with the messages. You usually indicate the loglevel with a macro. The loglevel macro expands to a string, which is concatenated to the message text at compile time; that's why there is no comma between the priority and the format string. Here are two examples of printk commands, a debug message and a critical message:

```
printk(KERN_DEBUG "Here I am: %s:%i\n", __FILE__, __LINE__);
printk(KERN_CRIT "I'm trashed; giving up on %p\n", ptr);
```

There are eight possible loglevel strings, defined in the header <linux/kernel.h>; we list them in order of decreasing severity:

KERN_EMERG
: Used for emergency messages, usually those that precede a crash.

KERN_ALERT
: A situation requiring immediate action.

KERN_CRIT
: Critical conditions, often related to serious hardware or software failures.

KERN_ERR
: Used to report error conditions; device drivers often use KERN_ERR to report hardware difficulties.

KERN_WARNING
: Warnings about problematic situations that do not, in themselves, create serious problems with the system.

KERN_NOTICE
: Situations that are normal, but still worthy of note. A number of security-related conditions are reported at this level.

KERN_INFO
: Informational messages. Many drivers print information about the hardware

they find at startup time at this level.

KERN_DEBUG
: Used for debugging messages. Each string (in the macro expansion) represents an integer in angle brackets. Integers range from 0 to 7, with smaller values representing higher priorities.

A printk statement with no specified priority defaults to DEFAULT_MESSAGE_LOGLEVEL, specified in kernel/printk.c as an integer. In the 2.6.10 kernel, DEFAULT_MESSAGE_LOGLEVEL is KERN_WARNING, but that has been known to change in the past.

Based on the loglevel, the kernel may print the message to the current console, be it a text-mode terminal, a serial port, or a parallel printer. If the priority is less than the integer variable console_loglevel, the message is delivered to the console one line at a time (nothing is sent unless a trailing newline is provided). If both klogd and syslogd are running on the system, kernel messages are appended to /var/log/messages independent of console_loglevel.

The variable console_loglevel is initialized to DEFAULT_CONSOLE_LOGLEVEL and can be modified through the sys_syslog system call. One way to change it is by specifying the –c switch when invoking klogd, as specified in the klogd manpage. Note that to change the current value, you must first kill klogd and then restart it with the –c option. Alternatively, you can write a program to change the console loglevel. The new level is specified as an integer value between 1 and 8, inclusive. If it is set to 1, only messages of level 0 (KERN_EMERG) reach the console; if it is set to 8, all messages, including debugging ones, are displayed.

It is also possible to read and modify the console loglevel using the text file /proc/sys/kernel/printk. The file hosts four integer values: the current loglevel, the default level for messages that lack an explicit loglevel, the minimum allowed loglevel, and the boot-time default loglevel. Writing a single value to this file changes the current loglevel to that value; thus, for example, you can cause all kernel messages to appear at the console by simply entering:

```
# echo 8 > /proc/sys/kernel/printk
```


Debugging Techniques

Redirecting Console Messages

Linux allows for some flexibility in console logging policies by letting you send messages to a specific virtual console. By default, the “console” is the current virtual terminal. To select a different virtual terminal to receive messages, you can issue ioctl(TIOCLINUX) on any console device. The following program, setconsole, can be used to choose which console receives kernel messages; it must be run by the superuser and is available in the misc-progs directory.

The following is the program in its entirety. You should invoke it with a single argument specifying the number of the console that is to receive messages.

```
int main(int argc, char **argv)
{
    char bytes[2] = {11,0}; /* 11 is the TIOCLINUX cmd number */
    if (argc= =2)
        bytes[1] = atoi(argv[1]); /* the chosen console */
    else
    {
        fprintf(stderr, "%s: need a single arg\n",argv[0]); exit(1);
    }
    if (ioctl(STDIN_FILENO, TIOCLINUX, bytes)<0)
    {
        /* use stdin */
        fprintf(stderr,"%s: ioctl(stdin, TIOCLINUX): %s\n",
        argv[0], strerror(errno));
        exit(1);
    }
    exit(0);
}
```

setconsole uses the special ioctl command TIOCLINUX, which implements Linux specific functions. To use TIOCLINUX, you pass it an argument that is a pointer to a byte array. The first byte of the array is a number that specifies the requested sub-command, and the following bytes are subcommand specific. In setconsole, subcommand 11 is used, and the next byte (stored in bytes[1]) identifies the virtual console. The complete description of TIOCLINUX can be found in drivers/char/tty_io.c, in the kernel

EmbLogic™ Embedded Technologies Pvt Ltd

sources.

Working with printk

Kernel programming brings its own, unique debugging challenges. Kernel code cannot be easily executed under a debugger, nor can it be easily traced, because it is a set of functionalities not related to any specific process. Kernel code errors can also be exceedingly hard to reproduce and can bring down the entire system with them, thus destroying much of the evidence that could be used to track them down.

printk

The most common debugging technique is monitoring using printk. printk classifies messages according to their severity by associating different loglevels, or priorities, with the messages. You usually indicate the loglevel with a macro. The loglevel macro expands to a string, which is concatenated to the message text at compile time; that's why there is no comma between the priority and the format string. Here are two examples of printk commands, a debug message and a critical message:

```
printk(KERN_DEBUG "Here I am: %s:%i\n", __FILE__, __LINE__);
printk(KERN_CRIT "I'm trashed; giving up on %p\n", ptr);
```

There are eight possible loglevel strings, defined in the header <linux/kernel.h>; we
list them in order of decreasing severity:

KERN_EMERG
: Used for emergency messages, usually those that precede a crash.

KERN_ALERT
: A situation requiring immediate action.

KERN_CRIT
: Critical conditions, often related to serious hardware or software failures.

KERN_ERR
: Used to report error conditions; device drivers often use KERN_ERR to report hardware difficulties.

KERN_WARNING
: Warnings about problematic situations that do not, in themselves, create serious problems with the system.

KERN_NOTICE
Situations that are normal, but still worthy of note. A number of security- related conditions are reported at this level.
KERN_INFO
Informational messages. Many drivers print information about the hardware they find at startup time at this level.
KERN_DEBUG
Used for debugging messages. Each string (in the macro expansion) represents an integer in angle brackets. Integers range from 0 to 7, with smaller values representing higher priorities.

A printk statement with no specified priority defaults to DEFAULT_MESSAGE_LOGLEVEL, specified in kernel/printk.c as an integer. In the 2.6.10 kernel, DEFAULT_MESSAGE_LOGLEVEL is KERN_WARNING, but that has been known to change in the past.

Based on the loglevel, the kernel may print the message to the current console, be it a text-mode terminal, a serial port, or a parallel printer. If the priority is less than the integer variable console_loglevel, the message is delivered to the console one line at a time (nothing is sent unless a trailing newline is provided). If both klogd and syslogd are running on the system, kernel messages are appended to /var/log/messages independent of console_loglevel.

The variable console_loglevel is initialized to DEFAULT_CONSOLE_LOGLEVEL and can be modified through the sys_syslog system call. One way to change it is by specifying the –c switch when invoking klogd, as specified in the klogd manpage. Note that to change the current value, you must first kill klogd and then restart it with the –c option. Alternatively, you can write a program to change the console loglevel. The new level is specified as an integer value between 1 and 8, inclusive. If it is set to 1, only messages of level 0 (KERN_EMERG) reach the console; if it is set to 8, all messages, including debugging ones, are displayed.

It is also possible to read and modify the console loglevel using the text file /proc/sys/kernel/printk. The file hosts four integer values: the current loglevel, the default level for messages that lack an explicit loglevel, the minimum allowed loglevel, and the boot-time default loglevel. Writing a single value to this file

changes the current loglevel to that value; thus, for example, you can cause all kernel messages to appear at the console by simply entering:

```
# echo 8 > /proc/sys/kernel/printk
```

Logging Messages

The printk function writes messages into a circular buffer that is __LOG_BUF_LEN bytes long: a value from 4 KB to 1 MB chosen while configuring the kernel. The dmesg command can be used to look at the content of the buffer without flushing it; actually, the command returns to stdout the whole content of the buffer, whether or not it has already been read.

If the circular buffer fills up, printk wraps around and starts adding new data to the beginning of the buffer, overwriting the oldest data. Therefore, the logging process loses the oldest data. This problem is negligible compared with the advantages of using such a circular buffer. For example, a circular buffer allows the system to run even without a logging process, while minimizing memory waste by overwriting old data should nobody read it. Another feature of the Linux approach to messaging is that printk can be invoked from anywhere, even from an interrupt handler, with no limit on how much data can be printed. The only disadvantage is the possibility of losing some data.

Printing Device Numbers

Occasionally, when printing a message from a driver, you will want to print the device number associated withp the hardware of interest. The kernel provides a couple of utility macros (defined in <linux/kdev_t.h>) for this purpose:

```
int print_dev_t(char *buffer, dev_t dev);
char *format_dev_t(char *buffer, dev_t dev);
```

Both macros encode the device number into the given buffer; the only difference is that print_dev_t returns the number of characters printed, while format_dev_t returns buffer; therefore, it can be used as a parameter to a printk call directly,

although one must remember that printk doesn't flush until a trailing newline is provided. The buffer should be large enough to hold a device number; given that 64-bit device numbers are a distinct possibility in future kernel releases, the buffer should probably be at least 20 bytes long.

Open And Release Methods

1.1: The open Method

The open method is provided for a driver to do any initialization in preparation for later operations. In most drivers, open should perform the following tasks:

- Check for device-specific errors (such as device-not-ready or similar hardware problems)
- Initialize the device if it is being opened for the first time
- Update the f_op pointer, if necessary
- Allocate and fill any data structure to be put in filp->private_data

The first order of business, however, is to identify which device is being opened. The prototype for the open method is:

```
int (*open)(struct inode *inode, struct file *filp);
```

The inode argument has the information we need in the form of its i_cdev field, which contains the cdev structure we set up before. The only problem is that we do not normally want the cdev structure itself, we want the scull_dev structure that contains that cdev structure.

The container_of macro, defined in *<linux/kernel.h>*:

```
container_of(pointer, container_type, container_field);
```

This macro takes a pointer to a field of type container_field, within a structure of type container_type, and returns a pointer to the containing structure. In scull_open, this macro is used to find the appropriate device structure:

```
struct scull_dev *dev; /* device information */

dev = container_of(inode->i_cdev, struct scull_dev, cdev);
```

```
filp->private_data = dev; /* for other methods */
```

Once it has found the scull_dev structure, scull stores a pointer to it in the private_data field of the file structure for easier access in the future.

The (slightly simplified) code for scull_open is:

```
int scull_open(struct inode *inode, struct file *filp)
{
    struct scull_dev *dev; /* device information */
    dev = container_of(inode->i_cdev, struct scull_dev, cdev);
    filp->private_data = dev; /* for other methods */
    /* now trim to 0 the length of the device if open was write-only */
    if ((filp->f_flags & O_ACCMODE) == O_WRONLY)
      {
                scull_trim(dev); /* ignore errors */
      }
    return 0;          /* success */
}
```

The scull device is global and persistent by design. Specifically, there's no action such as "initializing the device on first open," because we don't keep an open count for sculls.

The only real operation performed on the device is truncating it to a length of 0 when the device is opened for writing. This is performed because, by design, overwriting a scull device with a shorter file results in a shorter device data area.

1.2: The release Method

The role of the release method is the reverse of open. The device method should perform the following tasks:

- Deallocate anything that open allocated in filp->private_data
- Shut down the device on last close


```
int scull_release(struct inode *inode, struct file *filp)
{
    return 0;
}
```

What happens when a device file is closed more times than it is opened. After all, the dup and fork system calls create copies of open files without calling open; each of those copies is then closed at program termination. For example, most programs don't open their stdin file (or device), but all of them end up closing it. How does a driver know when an open device file has really been closed?

The answer is simple: not every close system call causes the release method to be invoked. Only the calls that actually release the device data structure invoke the method—hence its name. The kernel keeps a counter of how many times a file structure is being used. Neither fork nor dup creates a new file structure (only open does that); they just increment the counter in the existing structure. The close system call executes the release method only when the counter for the file structure drops to 0, which happens when the structure is destroyed. This relationship between the release method and the close system call guarantees that your driver sees only one release call for each open.

Note that the flush method is called every time an application calls close. However, very few drivers implement flush, because usually there's nothing to perform at close time unless release is involved.

As you may imagine, the previous discussion applies even when the application terminates without explicitly closing its open files: the kernel automatically closes any file at process exit time by internally using the close system call.

Open And Release Methods

The open Method

The open method is provided for a driver to do any initialization in preparation for later operations. In most drivers, open should perform the following tasks:

- Check for device-specific errors (such as device-not-ready or similar hardware problems)
- Initialize the device if it is being opened for the first time
- Update the f_op pointer, if necessary
- Allocate and fill any data structure to be put in filp->private_data

The first order of business, however, is to identify which device is being opened. The prototype for the open method is:

```
int (*open)(struct inode *inode, struct file *filp);
```

The inode argument has the information we need in the form of its i_cdev field, which contains the cdev structure we set up before. The only problem is that we do not normally want the cdev structure itself, we want the scull_dev structure that contains that cdev structure.

The container_of macro, defined in *<linux/kernel.h>*:

```
container_of(pointer, container_type, container_field);
```

This macro takes a pointer to a field of type container_field, within a structure of type container_type, and returns a pointer to the containing structure. In scull_open, this macro is used to find the appropriate device structure:

```
struct scull_dev *dev; /* device information */

dev = container_of(inode->i_cdev, struct scull_dev, cdev);
```

```
filp->private_data = dev; /* for other methods */
```

Once it has found the scull_dev structure, scull stores a pointer to it in the private_data field of the file structure for easier access in the future.

The (slightly simplified) code for scull_open is:

```
int scull_open(struct inode *inode, struct file *filp)
{
    struct scull_dev *dev; /* device information */
    dev = container_of(inode->i_cdev, struct scull_dev, cdev);
    filp->private_data = dev; /* for other methods */
    /* now trim to 0 the length of the device if open was write-only */
    if ((filp->f_flags & O_ACCMODE) == O_WRONLY)
      {
                scull_trim(dev); /* ignore errors */
      }
    return 0;          /* success */
}
```

The scull device is global and persistent by design. Specifically, there's no action such as "initializing the device on first open," because we don't keep an open count for sculls.

The only real operation performed on the device is truncating it to a length of 0 when the device is opened for writing. This is performed because, by design, overwriting a scull device with a shorter file results in a shorter device data area.

The release Method

The role of the release method is the reverse of open. The device method should perform the following tasks:

- Deallocate anything that open allocated in filp->private_data
- Shut down the device on last close


```
int scull_release(struct inode *inode, struct file *filp)
{
    return 0;
}
```

What happens when a device file is closed more times than it is opened. After all, the dup and fork system calls create copies of open files without calling open; each of those copies is then closed at program termination. For example, most programs don't open their stdin file (or device), but all of them end up closing it. How does a driver know when an open device file has really been closed?

The answer is simple: not every close system call causes the release method to be invoked. Only the calls that actually release the device data structure invoke the method—hence its name. The kernel keeps a counter of how many times a file structure is being used. Neither fork nor dup creates a new file structure (only open does that); they just increment the counter in the existing structure. The close system call executes the release method only when the counter for the file structure drops to 0, which happens when the structure is destroyed. This relationship between the release method and the close system call guarantees that your driver sees only one release call for each open.

Note that the flush method is called every time an application calls close. However, very few drivers implement flush, because usually there's nothing to perform at close time unless release is involved.

As you may imagine, the previous discussion applies even when the application terminates without explicitly closing its open files: the kernel automatically closes any file at process exit time by internally using the close system call.

Introduction to SCULL

scull's Memory Usage

How and why scull performs memory allocation. "How" is needed to thoroughly understand the code, and "why" demonstrates the kind of choices a driver writer needs to make, although scull is definitely not typical as a device.

We deals only with the memory allocation policy in scull and doesn't show the hardware management skills.

The region of memory used by scull, also called a device, is variable in length. The more you write, the more it grows; trimming is performed by overwriting the device with a shorter file.

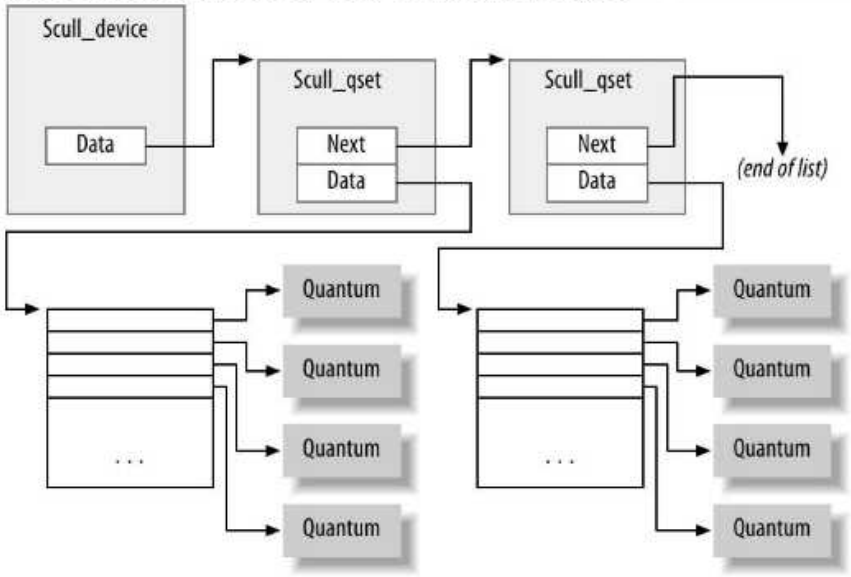
The scull driver introduces two core functions used to manage memory in the Linux kernel. These functions, defined in <linux/slab.h>, are:

```
void *kmalloc(size_t size, int flags);
void kfree(void *ptr);
```

A call to kmalloc attempts to allocate size bytes of memory; the return value is a pointer to that memory or NULL if the allocation fails. The flags argument is used to describe how the memory should be allocated. For now, we always use GFP_KERNEL. Allocated memory should be freed with kfree. You should never pass anything to kfree that was not obtained from kmalloc. It is, however, legal to pass a NULL pointer to kfree.

The source code for a smart implementation would be more difficult to read, and the aim of this section is to show read and write, not memory management. That's why the code just uses kmalloc and kfree without resorting to allocation of whole pages, although that approach would be more efficient. On the flip side, we didn't want to limit the size of the "device" area, for both a philosophical reason and a practical one. Philosophically, it's always a bad idea to put arbitrary limits on data items being managed. Practically, scull can be used to temporarily eat up your system's memory in order to run tests under low-memory conditions. Running such tests might help you understand the system's internals.

In scull, each device is a linked list of pointers, each of which points to a scull_dev structure. Each such structure can refer, by default, to at most four million bytes, through an array of intermediate pointers. The released source uses an array of 1000 pointers to areas of 4000 bytes. We call each memory area a quantum and the array (or its length) a quantum set. A scull device and its memory areas are shown in Figure.



The chosen numbers are such that writing a single byte in scull consumes 8000 or 12,000 thousand bytes of memory: 4000 for the quantum and 4000 or 8000 for the quantum set (according to whether a pointer is represented in 32 bits or 64 bits on the target platform). If, instead, you write a huge amount of data, the overhead of the linked list is not too bad. There is only one list element for every four megabytes of data, and the maximum size of the device is limited by the computer's memory size.

Choosing the appropriate values for the quantum and the quantum set is a

Introduction to SCULL

question of policy, rather than mechanism, and the optimal sizes depend on how the device is used. Thus, the scull driver should not force the use of any particular values for the quantum and quantum set sizes. In scull, the user can change the values in charge in several ways: by changing the macros SCULL_QUANTUM and SCULL_QSET in scull.h at compile time, by setting the integer values scull_quantum and scull_qset at module load time, or by changing both the current and default values using ioctl at runtime.

Using a macro and an integer value to allow both compile-time and load-time configuration is reminiscent of how the major number is selected. We use this technique for whatever value in the driver is arbitrary or related to policy. The only question left is how the default numbers have been chosen. In this particular case, the problem is finding the best balance between the waste of memory resulting from half-filled quanta and quantum sets and the overhead of allocation, deallocation, and pointer chaining that occurs if quanta and sets are small. Additionally, the internal design of kmalloc should be taken into account. The choice of default numbers comes from the assumption that massive amounts of data are likely to be written to scull while testing it, although normal use of the device will most likely transfer just a few kilobytes of data.

We have already seen the scull_dev structure that represents our device internally. That structure's quantum and qset fields hold the device's quantum and quantum set sizes, respectively. The actual data, however, is tracked by a different structure, which we call struct scull_qset:

```
struct scull_qset {
void **data;
struct scull_qset *next;
};
```

The next code fragment shows in practice how struct scull_dev and struct scull_qset are used to hold data. The function scull_trim is in charge of freeing the whole data area and is invoked by scull_open when the file is opened for writing. It simply walks through the list and frees any quantum and quantum set it finds.

EmbLogic Embedded Technologies Pvt Ltd

```
int scull_trim(struct scull_dev *dev)
{
     struct scull_qset *next, *dptr;
     int qset = dev->qset;
     /* "dev" is not-null */
     int i;
     for (dptr = dev->data; dptr; dptr = next)
     { /* all the list items */
          if (dptr->data)
          {
               for (i = 0; i < qset; i++)
               kfree(dptr->data[i]);
               kfree(dptr->data);
               dptr->data = NULL;
          }
          next = dptr->next;
          kfree(dptr);
     }
     dev->size = 0;
     dev->quantum = scull_quantum;
     dev->qset = scull_qset;
     dev->data = NULL;
     return 0;
}
```

scull_trim is also used in the module cleanup function to return memory used by scull to the system.

Operations

Read and Write

1: Read and write

The read and write methods both perform a similar task, that is, copying data from and to application code. Therefore, their prototypes are pretty similar:

```
ssize_t read(struct file *filp,
             char _ _user *buff,
             size_t count,
             loff_t *offp);

ssize_t write(struct file *filp,
              const char _ _user *buff,
              size_t count, loff_t *offp);
```

For both methods, filp is the file pointer and count is the size of the requested data transfer. The buff argument points to the user buffer holding the data to be written or the empty buffer where the newly read data should be placed. Finally, offp is a pointer to a "long offset type" object that indicates the file position the user is accessing. The return value is a "signed size type"; its use is discussed later.

The buff argument to the read and write methods is a user-space pointer. Therefore, it cannot be directly dereferenced by kernel code. There are a few reasons for this restriction:

- Depending on which architecture your driver is running on, and how the kernel was configured, the user-space pointer may not be valid while running in kernel mode at all. There may be no mapping for that address, or it could point to some other, random data.
- Even if the pointer does mean the same thing in kernel space, user-space memory is paged, and the memory in question might not be resident in RAM when the system call is made. Attempting to reference the user-space memory directly could generate a page fault, which is something that kernel code is not allowed to do. The result would be an "oops," which would result in the death of the process that made the system call.
- The pointer in question has been supplied by a user program, which could be buggy or malicious. If your driver ever blindly dereferences a user-supplied pointer, it provides an open doorway allowing a user-space program to access or overwrite memory anywhere in the system. If you do not wish to be responsible for compromising the security of your users' systems, you cannot ever dereference a user-space pointer directly.

Obviously, your driver must be able to access the user-space buffer in order to get its job done. This access must always be performed by special, kernel-supplied functions. We introduce some of those functions (which are defined in *<asm/uaccess.h>*).

The code for read and write in scull needs to copy a whole segment of data to or from the user address space. This capability is offered by the following kernel functions, which copy an arbitrary array of bytes and sit at the heart of most read and write implementations:

```
unsigned long copy_to_user(void __user *to,
                           const void *from,
                           unsigned long count);

unsigned long copy_from_user(void *to,
                             const void __user *from,
                             unsigned long count);
```

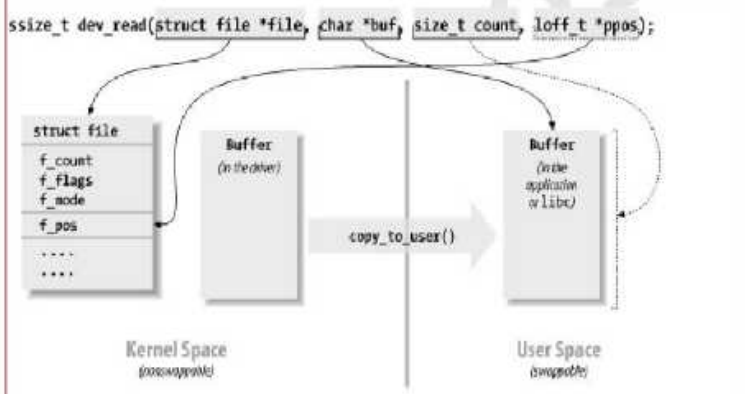
The role of the two functions is not limited to copying data to and from user-space: they also check whether the user space pointer is valid. If the pointer is invalid, no copy is performed; if an invalid address is encountered during the copy, on the other hand, only part of the data is copied. In both cases, the return value is the amount of memory still to be copied. The scull code looks for this error return, and returns -EFAULT to the user if it's not 0.

As far as the actual device methods are concerned, the task of the read method is to copy data from the device to user space (using copy_to_user), while the write method must copy data from user space to the device (using copy_from_user). Each read or write system call requests transfer of a specific number of bytes, but the driver is free to transfer less data.

Whatever the amount of data the methods transfer, they should generally update the file position at *offp to represent the current file position after successful completion of the system call. The kernel then propagates the file position change back into the file structure when appropriate.

Figure: represents how a typical read implementation uses its arguments.

Figure: The arguments to read



Both the read and write methods return a negative value if an error occurs. A return value greater than or equal to 0, instead, tells the calling program how many bytes have been successfully transferred. If some data is transferred correctly and then an

error happens, the return value must be the count of bytes successfully transferred, and the error does not get reported until the next time the function is called. Implementing this convention requires, of course, that your driver remember that the error has occurred so that it can return the error status in the future.

Although kernel functions return a negative number to signal an error, and the value of the number indicates the kind of error that occurred, programs that run in user space always see -1 as the error return value. They need to access the errno variable to find out what happened. The user-space behavior is dictated by the POSIX standard, but that standard does not make requirements on how the kernel operates internally.

The read Method

The return value for read is interpreted by the calling application program:

- If the value equals the count argument passed to the read system call, the requested number of bytes has been transferred. This is the optimal case.
- If the value is positive, but smaller than count, only part of the data has been transferred. This may happen for a number of reasons, depending on the device. Most often, the application program retries the read. For instance, if you read using the fread function, the library function reissues

the system call until completion of the requested data transfer.

- If the value is 0, end-of-file was reached (and no data was read).
- A negative value means there was an error. The value specifies what the error was, according to *<linux/errno.h>*. Typical values returned on error include -EINTR (interrupted system call) or -EFAULT (bad address).

What is missing from the preceding list is the case of "there is no data, but it may arrive later." In this case, the read system call should block. We'll deal with blocking input later.

The scull code takes advantage of these rules. In particular, it takes advantage of the partial-read rule. Each invocation of scull_read deals only with a single data quantum, without implementing a loop to gather all the data; this makes the code shorter and easier to read. If the reading program really wants more data, it reiterates the call. If the standard I/O library (i.e., fread) is used to read the device, the application won't even notice the quantization of the data transfer.

If the current read position is greater than the device size, the read method of scull returns 0 to signal that there's no data available (in other words, we're at end-of-file). This situation can happen if process A is reading the device while process B opens it for writing, thus truncating the device to a length of 0.

Process A suddenly finds itself past end-of-file, and the next read call returns 0.

Here is the code for read (ignore the calls to down_interruptible and up for now; we will get to them later):

```
ssize_t scull_read(struct file *filp,
                   char _ _user *buf,
                   size_t count,
                   loff_t *f_pos)
{
    Declerations:
        get a local device representation.
        Set quantum and qset
        The first listitem
        Set how many bytes in the listitem
        Get semaphore
        Check if file position is greater then device size
        check if (*f_pos + count > dev->size)
            take appropriate action
        find listitem, qset index, and offset in the quantum
        follow the list up to the right position (scull_follow)
        don't fill holes
```

read only up to the end of this quantum

copy_to_user

update f_pos

release semaphore

The write Method

write, like read, can transfer less data than was requested, according to the following rules for the return value:

- If the value equals count, the requested number of bytes has been transferred.
- If the value is positive, but smaller than count, only part of the data has been transferred. The program will most likely retry writing the rest of the data.
- If the value is 0, nothing was written. This result is not an error, and there is no reason to return an error code. Once again, the standard library retries the call to write. We'll examine the exact meaning of this case later, where blocking write is introduced.
- A negative value means an error occurred; as for read, valid error values are those defined in *<linux/errno.h>*.

The scull code for write deals with a single quantum at a time, as the read method does:

Writing

The write Method

write, can transfer less data than was requested, according to the following rules for the return value:

- If the value equals count, the requested number of bytes has been transferred.
- If the value is positive, but smaller than count, only part of the data has been transferred. The program will most likely retry writing the rest of the data.
- If the value is 0, nothing was written. This result is not an error, and there is no reason to return an error code. Once again, the standard library retries the call to write.
- A negative value means an error occurred; as for read, valid error values are those defined in <linux/errno.h>.

Unfortunately, there may still be misbehaving programs that issue an error message and abort when a partial transfer is performed. This happens because some programmers are accustomed to seeing write calls that either fail or succeed completely, which is actually what happens most of the time and should be supported by devices as well.

The scull code for write deals with a single quantum at a time, as the read method does:

```
ssize_t scull_write(struct file *filp, const char __user *buf, size_t count,
loff_t *f_pos)
{
    struct scull_dev *dev = filp->private_data;
    struct scull_qset *dptr;
    int quantum = dev->quantum, qset = dev->qset;
    int itemsize = quantum * qset;
    int item, s_pos, q_pos, rest;
    ssize_t retval = -ENOMEM; /* value used in "goto out" statements */
    if (down_interruptible(&dev->sem))
        return -ERESTARTSYS;
    /* find listitem, qset index and offset in the quantum */
    item = (long)*f_pos / itemsize;
```

```
    rest = (long)*f_pos % itemsize;
    s_pos = rest / quantum; q_pos = rest % quantum;
    /* follow the list up to the right position */
    dptr = scull_follow(dev, item);
    if (dptr = = NULL)
    goto out;
    if (!dptr->data)
    {
        dptr->data = kmalloc(qset * sizeof(char *), GFP_KERNEL);
        if (!dptr->data)
        goto out;
        memset(dptr->data, 0, qset * sizeof(char *));
    }
    if (!dptr->data[s_pos])
    {
        dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
        if (!dptr->data[s_pos])
        goto out;
    }
    /* write only up to the end of this quantum */
    if (count > quantum - q_pos)
    count = quantum - q_pos;
    if (copy_from_user(dptr->data[s_pos]+q_pos, buf, count))
    {
        retval = -EFAULT;
        goto out;
    }
    *f_pos += count;
    retval = count;
    /* update the size */
    if (dev->size < *f_pos)
    dev->size = *f_pos;
out:
    up(&dev->sem);
    return retval;
}
```

Vectored Read Write

1.1: readv and writev

These "vector" versions of read and write take an array of structures, each of which contains a pointer to a buffer and a length value. A readv call would then be expected to read the indicated amount into each buffer in turn. writev, instead, would gather together the contents of each buffer and put them out as a single write operation.

If your driver does not supply methods to handle the vector operations, readv and writev are implemented with multiple calls to your read and write methods.

The prototypes for the vector operations are:

```
ssize_t (*readv) (struct file *filp, const struct iovec *iov, unsigned long count, loff_t *ppos);

ssize_t (*writev) (struct file *filp, const struct iovec *iov,unsigned long count, loff_t *ppos);
```

Here, the filp and ppos arguments are the same as for read and write. The iovec structure, defined in *<linux/uio.h>*, looks like:

```
struct iovec
{
    void _  _user *iov_base;
    _ _kernel_size_t iov_len;
};
```

Each iovec describes one chunk of data to be transferred; it starts at iov_base (in user space) and is iov_len bytes long. The count parameter tells the method how many iovec structures there are. These structures are created by the application, but the kernel copies them into kernel space before calling the driver.

The simplest implementation of the vectored operations would be a straightforward loop that just passes the address and length out of each iovec to the driver's read or write function.

Many drivers, gain no benefit from implementing these methods themselves. Therefore, scull omits them. The kernel emulates them with read and write, and the end result is the same.

1.2:. Playing with the New Devices

Once you are equipped with the methods just described, the driver can be compiled and tested; it retains any data you write to it until you overwrite it with new data. The device acts like a data buffer whose length is limited only by the amount of real RAM available.

The free command can be used to see how the amount of free memory shrinks and expands according to how much data is written into scull.

To get more confident with reading and writing one quantum at a time, you can add a printk at an appropriate point in the driver and watch what happens while an application reads or writes large chunks of data. Alternatively, use the strace utility to monitor the system calls issued by a program, together with their return values. Tracing a cp or an ls -l > /dev/scull0 shows quantized reads and writes.

Read And Write Methods

1.1: Read and write

The read and write methods both perform a similar task, that is, copying data from and to application code. Therefore, their prototypes are pretty similar:

```
ssize_t read(struct file *filp, char __user *buff, size_t count, loff_t *offp);

ssize_t write(struct file *filp, const char __user *buff, size_t count, loff_t *offp);
```

For both methods, filp is the file pointer and count is the size of the requested data transfer. The buff argument points to the user buffer holding the data to be written or the empty buffer where the newly read data should be placed. Finally, offp is a pointer to a "long offset type" object that indicates the file position the user is accessing. The return value is a "signed size type"; its use is discussed later.

The buff argument to the read and write methods is a user-space pointer. Therefore, it cannot be directly dereferenced by kernel code. There are a few reasons for this restriction:

- Depending on which architecture your driver is running on, and how the kernel was configured, the user-space pointer may not be valid while running in kernel mode at all. There may be no mapping for that address, or it could point to some other, random data.
- Even if the pointer does mean the same thing in kernel space, user-space memory is paged, and the memory in question might not be resident in RAM when the system call is made. Attempting to reference the user-space memory directly could generate a page fault, which is something that kernel code is not allowed to do. The result would be an "oops," which would result in the death of the process that made the system call.
- The pointer in question has been supplied by a user program, which could be buggy or malicious. If your driver ever blindly dereferences a user-supplied pointer, it provides an open doorway allowing a user-space

program to access or overwrite memory anywhere in the system. If you do not wish to be responsible for compromising the security of your users' systems, you cannot ever dereference a user-space pointer directly.

Obviously, your driver must be able to access the user-space buffer in order to get its job done. This access must always be performed by special, kernel-supplied functions. We introduce some of those functions (which are defined in *<asm/uaccess.h>*).

The code for read and write in scull needs to copy a whole segment of data to or from the user address space. This capability is offered by the following kernel functions, which copy an arbitrary array of bytes and sit at the heart of most read and write implementations:

unsigned long copy_to_user(void _ _user *to, const void *from, unsigned long count);

unsigned long copy_from_user(void *to, const void _ _user *from, unsigned long count);

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 scull code looks for this error return, and returns -EFAULT to the user if it's not 0.

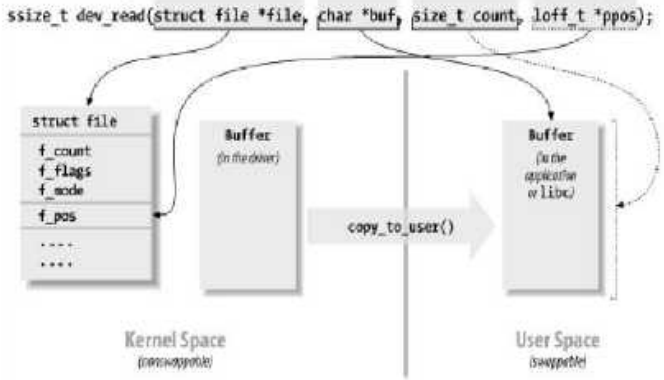
As far as the actual device methods are concerned, the task of the read method is to copy data from the device to user space (using copy_to_user), while the write method must copy data from user space to the device (using copy_from_user). Each read or write system call requests transfer of a specific number of bytes, but the driver is free to transfer less data.

Whatever the amount of data the methods transfer, they should generally

update the file position at *offp to represent the current file position after successful completion of the system call. The kernel then propagates the file position change back into the file structure when appropriate.

Figure -2 represents how a typical read implementation uses its arguments.

Figure -2. The arguments to read



Both the read and write methods return a negative value if an error occurs. A return value greater than or equal to 0, instead, tells the calling program how many bytes have been successfully transferred. If some data is transferred correctly and then an error happens, the return value must be the count of bytes successfully transferred, and the error does not get reported until the next time the function is called. Implementing this convention requires, of course, that your driver remember that the error has occurred so that it can return the error status in the future.

Although kernel functions return a negative number to signal an error,

and the value of the number indicates the kind of error that occurred, programs that run in user space always see -1 as the error return value. They need to access the errno variable to find out what happened. The user-space behavior is dictated by the POSIX standard, but that standard does not make requirements on how the kernel operates internally.

1.2: The read Method

The return value for read is interpreted by the calling application program:

- If the value equals the count argument passed to the read system call, the requested number of bytes has been transferred. This is the optimal case.
- If the value is positive, but smaller than count, only part of the data has been transferred. This may happen for a number of reasons, depending on the device. Most often, the application program retries the read. For instance, if you read using the fread function, the library function reissues the system call until completion of the requested data transfer.
- If the value is 0, end-of-file was reached (and no data was read).
- A negative value means there was an error. The value specifies what the error was, according to *<linux/errno.h>*. Typical values returned on error include -EINTR (interrupted system call) or -EFAULT (bad address).

What is missing from the preceding list is the case of "there is no data, but it may arrive later." In this case, the read system call should block. We'll deal with blocking input later.

The scull code takes advantage of these rules. In particular, it takes advantage of the partial-read rule. Each invocation of scull_read deals only with a single data quantum, without implementing a loop to gather all the data; this makes the code shorter and easier to read. If the reading program really wants more data, it reiterates the call. If the standard I/O library (i.e., fread) is used to read the device, the application won't even notice the quantization of the data transfer.

If the current read position is greater than the device size, the read method of scull returns 0 to signal that there's no data available (in other words, we're at end-of-file). This situation can happen if process A is reading the device while process B opens it for writing, thus truncating the device to a length of 0. Process A suddenly finds itself past end-of-file, and the next read call returns 0.

Here is the code for read (ignore the calls to down_interruptible and up for now; we will get to them later):

```
ssize_t scull_read(struct file *filp, char __user *buf, size_t count, loff_t
*f_pos)
{
    struct scull_dev *dev = filp->private_data;
    struct scull_qset *dptr;                /* the first listitem */
    int quantum = dev->quantum;
    int qset = dev->qset;
    int itemsize = quantum * qset;                  /*how many bytes in the
listitem */
    int item, s_pos, q_pos, rest;
    ssize_t retval = 0;
    if (down_interruptible(&dev->sem))
        return -ERESTARTSYS;
    if (*f_pos >= dev->size)
        goto out;
    if (*f_pos + count > dev->size)
        count = dev->size - *f_pos;
    /* find listitem, qset index, and offset in the quantum */
    item = (long)*f_pos / itemsize;
    rest = (long)*f_pos % itemsize;
    s_pos = rest / quantum;
    q_pos = rest % quantum;
    /* follow the list up to the right position (defined elsewhere) */
    dptr = scull_follow(dev, item);
    if (dptr == NULL || !dptr->data || ! dptr->data[s_pos])
        goto out; /* don't fill holes */
    /* read only up to the end of this quantum */
    if (count > quantum - q_pos)
```

```
        count = quantum - q_pos;
    if (copy_to_user(buf, dptr->data[s_pos] + q_pos, count)) {
        retval = -EFAULT;
        goto out;
    }
    *f_pos += count;
    retval = count;
  out:
    up(&dev->sem);
    return retval;
}
```

12: The write Method

write, like read, can transfer less data than was requested, according to the following rules for the return value:

- If the value equals count, the requested number of bytes has been transferred.
- If the value is positive, but smaller than count, only part of the data has been transferred. The program will most likely retry writing the rest of the data.
- If the value is 0, nothing was written. This result is not an error, and there is no reason to return an error code. Once again, the standard library retries the call to write. We'll examine the exact meaning of this case later, where blocking write is introduced.
- A negative value means an error occurred; as for read, valid error values are those defined in *<linux/errno.h>*.

The scull code for write deals with a single quantum at a time, as the read method does:

```
ssize_t scull_write(struct file *filp, const char _ _user *buf, size_t count,
loff_t *f_pos)
{
    struct scull_dev *dev = filp->private_data;
    struct scull_qset *dptr;
```



```
int quantum = dev->quantum, qset = dev->qset;
int itemsize = quantum * qset;
int item, s_pos, q_pos, rest;
ssize_t retval = -ENOMEM; /* value used in "goto out" statements */
if (down_interruptible(&dev->sem))
    return -ERESTARTSYS;
/* find listitem, qset index and offset in the quantum */
item = (long)*f_pos / itemsize;
rest = (long)*f_pos % itemsize;
s_pos = rest / quantum; q_pos = rest % quantum;
/* follow the list up to the right position */
dptr = scull_follow(dev, item);
if (dptr =  = NULL)
    goto out;
if (!dptr->data) {
    dptr->data = kmalloc(qset * sizeof(char *), GFP_KERNEL);
    if (!dptr->data)
        goto out;
    memset(dptr->data, 0, qset * sizeof(char *));
}
if (!dptr->data[s_pos]) {
    dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
    if (!dptr->data[s_pos])
        goto out;
}
/* write only up to the end of this quantum */
if (count > quantum - q_pos)
    count = quantum - q_pos;
if (copy_from_user(dptr->data[s_pos]+q_pos, buf, count)) {
    retval = -EFAULT;
    goto out;
}
*f_pos += count;
retval = count;
    /* update the size */
if (dev->size < *f_pos)
    dev->size = *f_pos;
out:
```

```
    up(&dev->sem);
    return retval;
}
```