

Table of Contents

Chapter 1: User-Kernel Communication Pathways	1
Your bookmark	1
Technical requirements	2
Approaches to communicating/interfacing a kernel module with a userspace C app	2
Interfacing via the proc filesystem (procfs)	3
Understanding the proc filesystem	3
Directories under /proc	4
The purpose behind the proc filesystem	5
procfs is off-bounds to driver authors	5
Using procfs to interface with userspace	6
Basic procfs APIs	6
The four procfs files we create	8
Trying out the dynamic debug_level procfs control	8
Dynamically controlling the debug_level via procfs	11
A few misc procfs APIs	15
Interfacing via the sys filesystem (sysfs)	16
Creating a sysfs (pseudo) file in code	17
Creating a simple platform device	18
Platform devices	18
Tying it all together - setting up the device attributes and creating the sysfs file	20
The code implementing our sysfs file creation and it's callbacks	22
The 'one value per sysfs file' rule	26
Interfacing via the debug filesystem (debugfs)	27
Checking for the presence of debugfs	28
Looking up the debugfs API documentation	29
An interfacing example with debugfs	29
Creating and using the first debugfs file	30
Creating and using the second debugfs file	33
Helper debugfs APIs for working on numeric globals	34
Removing the debugfs pseudo file(s)	35
Seeing a kernel bug - an Oops!	36
Debugfs - actual users	39
Interfacing via netlink sockets	40
Advantages using sockets	41
Understanding what a netlink socket is	41
Writing the userspace netlink socket application	42
Writing the kernel-space netlink socket code as a kernel module	45
Trying out our netlink interfacing project	47

Interfacing via the ioctl system call	49
Using the ioctl in user and kernel space	51
Userspace - using the ioctl system call	53
Kernel space - using the ioctl system call	54
The ioctl as a debug interface	57
Comparing the interfacing methods - a table	57
Summary	58
Questions	59
Further Reading	60
Index	62

1

User-Kernel Communication Pathways

Consider this scenario: you've successfully developed a device driver for, say, a pressure sensor device (by perhaps using the kernel's I2C APIs to fetch the pressure from the chip via the I2C protocol). So, you have the pressure value in a variable within the driver which of course implies that it's within kernel-space. The issue at hand: how exactly do you now have a *userspace application retrieve this value*? Well, as we learned in the previous chapter, you can always include a `.read` method in the driver's *fops* structure. When the userspace app issues a `read(2)` system call, control will be diverted (via the VFS) to your driver's *read method*. In there, you perform a `copy_to_user()` (or equivalent), resulting in the usermode app receiving the value. Yes, but there are other, sometimes superior, ways to do so.

In this chapter, we have you understand various available communication interfaces or pathways - means to communicate or interface between user and kernel address spaces. This is an important aspect of writing kernel/driver code, for without this knowledge, how will you be able to achieve a key thing - efficiently transfer information between a kernel-space component (often a device driver but it could be anything, really), and a userspace process or thread. Not only that, some of these techniques that we shall learn about, are quite often used for debug purposes as well. In this chapter, we cover several techniques to effect communication between the kernel and user (virtual) address spaces - communication via traditional *proc* filesystem - *procfs*, the better way for drivers via the 'sys' filesystem - *sysfs*, via a debug filesystem - *debugfs*, *netlink sockets* and the `ioctl(2)` system call.

The following topics will be covered in this chapter:

- Approaches to communicating/interfaces a kernel module with userspace C app
- Interfacing via the *proc* filesystem (*procfs*)
- Interfacing via the *sys* filesystem (*sysfs*)
- Interfacing via the debug filesystem (*debugfs*)

- Interfacing via netlink sockets
- Interfacing via the `ioctl` system call.
- Comparing the interfacing methods

Technical requirements

I assume the you have gone through Chapter 1 and have appropriately prepared a guest VM (Virtual Machine) running Ubuntu 18.04 LTS (or a later stable release) and installed all required packages. If not, I recommend you do this first.

To get the maximum out of this book, I strongly recommend you first set up the workspace environment including cloning this book's GitHub repository for the code, and work on it in a hands-on fashion.

Approaches to communicating/interfacing a kernel module with a userspace C app

As mentioned in the introduction, in this chapter we wish to learn how to efficiently transfer information between a kernel-space component (often a device driver but it could be anything, really), and a userspace process or thread. To begin, lets simply enumerate various techniques available to the kernel or driver author to communicate or interface with a userspace 'C' application. Well, the userspace component could be a C app, a shell script (both of which we typically show in this book), or even other apps like C++/Java apps, Python/Perl scripts or such others.

You will recall from the basics of Linux kernel architecture that we discussed in Chapter 4, *Writing your First Kernel Module - LKMs Part 1* under the subsection *Library and System Call APIs*, that the essential interface between user space applications and the kernel which includes device drivers are the **system call APIs**. Now, in the previous chapter, you learned the basics of writing a character device driver for Linux. Within that, you also learned how to transfer data between user and kernel address spaces by having a usermode application open the device file and issuing `read(2)` and `write(2)` system calls, resulting in the driver's read/write method being invoked by the VFS and the driver performing the data transfer via the `copy_{from|to}_user()` APIs. So, the question here is: if we have already covered that, then what else is there to learn about in this regard?

Ah, quite a bit! The reality is that there are several other techniques of interfacing between a usermode app and the kernel. Certainly, they all very much depend upon using system

calls; after all, there is no other way to enter the kernel! Nevertheless, the techniques differ. Our aim in this chapter is to show you various communication interfaces that are available, as of course depending on the project, one might be more suitable than others to use. Below, we mention various techniques described in this chapter to interface between user and kernel address spaces:

- via the traditional `procfs` interface
- via `sysfs`
- via `debugfs`
- via netlink sockets
- via the `ioctl(2)` system call

Throughout this chapter we will discuss these interfacing techniques in detail. In addition, we will also briefly explore how conducive they are to the purpose of *debugging*. So let's begin with using the `procfs` interface.

Interfacing via the `proc` filesystem (`procfs`)

In this section we shall cover what the `proc` filesystem is and how you can leverage it for usage as an interface between user and kernel address spaces. The `proc` filesystem is a powerful and easy-to-program interface, often used for status reporting and debug of core kernel systems.



Note that from version 2.6 Linux onward and for upstream contribution, this interface is *not* to be used by driver authors (it's strictly meant for kernel-internal usage only).

Understanding the `proc` filesystem

Linux has a virtual filesystem named *proc*; the default mount point for which is `/proc`. The first thing to realize regarding the `proc` filesystem is that its content is *not* on a non-volatile disk. Its content is in RAM, and is thus volatile. The files and directories one sees under `/proc` are pseudo-files setup by the kernel code for `proc`; the kernel hints at this fact by almost always showing the file *size* as zero:

```
$ mount |grep -w proc
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
$ ls -l /proc/
total 0
```

```

dr-xr-xr-x  8 root  root      0 Jan 27 11:13 1/
dr-xr-xr-x  8 root  root      0 Jan 29 08:22 10/
dr-xr-xr-x  8 root  root      0 Jan 29 08:22 11/
dr-xr-xr-x  8 root  root      0 Jan 29 08:22 11550/
[...]
-r--r--r--  1 root  root      0 Jan 29 08:22 consoles
-r--r--r--  1 root  root      0 Jan 29 08:19 cpuinfo
-r--r--r--  1 root  root      0 Jan 29 08:22 crypto
-r--r--r--  1 root  root      0 Jan 29 08:20 devices
-r--r--r--  1 root  root      0 Jan 29 08:22 diskstats
[...]
-r--r--r--  1 root  root      0 Jan 29 08:22 vmstat
-r--r--r--  1 root  root      0 Jan 29 08:22 zoneinfo
$

```

Below, we summarize a few critical points regarding Linux's powerful proc filesystem.



The objects under `/proc` (files, directories, soft links, and so on) are all *pseudo* objects; they live in RAM

Directories under `/proc`

The directories under `/proc` represent processes. The name of the directory is the PID of the process (technically, it's the TGID of the process. We covered this back in Chapter 6, Kernel and Memory Management Internals Essentials).

This folder contains information regarding this process. So, for example, for the *init* process (always PID 1), you can examine detailed information about this process (it's attributes, open files, memory layout, children, and so on) under the folder `/proc/1`. Recall though, that on most modern Linux distro's PID 1 represents the *systemd* process)

As an example, in the screenshot below, we gain a root shell and do an `ls /proc/1`:

```

$ sudo -i
root@llkd-vbox:~# ls /proc/1
arch_status  cpuset    limits    net        personality  smaps_rollup  timerslack_ns
autogroup    cwd       loginuid  ns          projid_map   stack         uid_map
auxv         environ  map_files numa_maps   root         stat          wchan
cgroup       exe       maps      oom_adj     sched        statm
clear_refs   fd        mem       oom_score   schedstat    status
cmdline      fdinfo    mountinfo oom_score_adj sessionid     syscall
comm         gid_map   mounts    pagemap     setgroups    task
coredump_filter io         mountstats patch_state  smaps        timers
root@llkd-vbox:~#

```

The complete details regarding the pseudo files and folders under `/proc/<PID>/...` can be found in the man page on `proc(5)` (by doing `man 5 proc`); do refer to it!

The precise content under `/proc` varies with both the kernel version and the (CPU) architecture; x86[-64] tends to have the richest content

The purpose behind the proc filesystem

The *purpose* behind the `proc` filesystem is two-fold:

- one, it is a simple interface for developers, sys admins, anyone really to look deep inside the kernel, and gain information regarding the internals of processes, the kernel and even hardware. Using this interface only requires you to know basic shell commands like `cd`, `cat`, `echo`, `ls`, and so on.
- two, as the *root* user and at times the owner, you can write into certain pseudo files under `/proc/sys`, thus *tuning* various kernel parameters. This feature is called *sysctl*. As an example, you can tune various IPv4 networking parameters in `/proc/sys/net/ipv4/`. They are all documented here: <https://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt>

Changing the value of a `proc`-based tunable is easy; for example, let's change the maximum number of threads allowed at any given point in time on the box. Run the below commands as *root* of course:

```
# cat /proc/sys/kernel/threads-max
15741
# echo 10000 > /proc/sys/kernel/threads-max
# cat /proc/sys/kernel/threads-max
10000
#
```

There, done. However, it should be clear that the above operation is *volatile* - the change only applies to this session; a power-cycle or reboot will result in it reverting back to the default value of course. How then, does one make the change *permenant*? The short answer: use the `sysctl(8)` utility to do so; refer to its man page for the details.

Ready to write `procfs`-interfacing code now? Not so fast, the next section informs you as to why this may *not* be a great idea after all.

procfs is off-bounds to driver authors

Even though we could use the `proc` filesystem to interface with a usermode app, there is an

important point to note here! You must realize that `procfs` is, like many similar facilities within the kernel, an **Application Binary Interface (ABI)**. The kernel community makes no promises that it remains stable and exactly the way it is today, just as is the case with the kernel *APIs* and its internal data structures as well. In fact, ever since the 2.6 kernel, the kernel folks have made this very clear- *device driver authors (and the like) are not supposed to use `procfs` for their own purposes, their interfaces, debug or otherwise*. Earlier to 2.6 Linux, it was quite common to use `proc` for said purposes (abused, as per the kernel community, as `proc` is meant for kernel internal use only!).

So, all right, if `procfs` is considered off-bounds, or deprecated, to us as driver authors, then what do we use? Driver authors are to use the `sysfs` facility to *export* their interfaces. In reality it's not just `sysfs`; there are several choices available to you such as the `sysfs`, `debugfs`, `netlink` sockets, the `ioctl` system call. We cover these in some detail later in this chapter.

Hang on though; again, the reality is that this 'rule' regarding the non-usage of `procfs` for driver authors are for the community. What is meant is: if you intend to *upstream* your driver or kernel module to the mainline kernel thus contributing your code under the GPLv2 license, *then* all the community rules apply. If not, it's really up to you to decide. Of course, following the kernel community's guidelines and rules can only be a good thing; we definitely recommend you do so. To discourage usage of `proc` by non-core stuff like drivers, there is no kernel documentation available for the `proc` API/ABI.

FYI, on the 5.4.0 kernel, there are around 70-odd callers of the `proc_create()` kernel API, several of them being (typically older) drivers and filesystems.

Nevertheless (you have been warned!), now let's move into the details of interfacing a userspace process with kernel code via `procfs`.

Using `procfs` to interface with userspace

As a kernel module or device driver developer, we can actually create our own entries under `/proc`, leveraging this as a simple interface to userspace. How can we do this? The kernel provides APIs to create directories and files under `procfs`, we learn to use them.

Basic `procfs` APIs

Here, we do not intend to delve into the gory details of the `procfs` API set; rather, we shall cover just enough to have you be able to understand and use them. For deeper detail, do refer the ultimate resource: the kernel codebase. The routines we cover below are exported, thus making them available to module/driver authors like you. Also, as mentioned earlier,

all the procfs file objects are really pseudo objects, in the sense that they exist only in RAM.

Lets begin to explore a few simple procfs APIs enabling you to perform a few key tasks - creating a directory under the proc filesystem, creating (pseudo) files under there, and deleting them, respectively. For all of these tasks, do ensure you include the relevant header file:

```
#include <linux/proc_fs.h>
```

1. Create a directory named *name* under */proc*:

```
struct proc_dir_entry *proc_mkdir(const char *name,
                                struct proc_dir_entry *parent);
```

The first parameter is the name of the directory, the second parameter is the pointer to the parent directory to create it under. Passing `NULL` here creates the directory under the root, that is, under */proc*. Save the return value, as you will typically use it as a parameter in subsequent APIs



FYI, the routine `proc_mkdir_data()` allows you to pass along a data item (a `void *`) as well; note though that it's exported via `EXPORT_SYMBOL_GPL`

2. Create a procfs (pseudo) 'file' */proc/parent/name*:

```
struct proc_dir_entry *proc_create(const char *name, umode_t mode,
                                struct proc_dir_entry *parent,
                                const struct file_operations *proc_fops);
```

The key parameter here is the `struct file_operations` that we introduced in *Chapter 11, Writing a simple Misc Character Device Driver*. You are expected to populate it with the 'methods' to be implemented (more on this below). Think about it, this is really powerful stuff: using the '*fops*' structure, we can setup 'callback' functions which the kernel's proc filesystem layer will honor: when a userspace process reads from our proc file, it will invoke the `.read` method or callback function, if a userspace app writes, it will invoke the `.write` callback!

3. Remove a procfs entry:

```
void remove_proc_entry(const char *name, struct proc_dir_entry
                      *parent)
```

This API removes the specified entry */proc/name* entry and frees it (if not in use); similarly (and often more convenient), use the `remove_proc_subtree()` API to

remove an entire sub-tree within */proc* (typically on cleanup and/or if an error occurs).

Now that we have the basics, the empirical approach demands that we put these APIs to practice! To do so, let's first figure out what directories/files to create under */proc*; the section below covers just this.

The four procfs files we create

To help clearly illustrate the usage of procfs as an interfacing technology, we have our kernel module create a directory under */proc* and, within that directory, four procfs (pseudo) files. Note that all procfs files by default have their *owner:group* attributes as *root:root*. Okay, now, we intend to do this: create a directory */proc/proc_simple_intf* and, under it, create four (pseudo) files: the names and attributes of the four procfs (pseudo) files under the */proc/proc_simple_intf* directory are shown in *Table 1* below:

<i>Name of procfs 'file'</i>	<i>R: action on read callback, invoked via userspace read</i>	<i>W: action on write callback, invoked via userspace write</i>	<i>Procfs 'file' permissions</i>
<code>llkdproc_dbg_level</code>	Retrieves (to userspace) the current value of the global var <code>debug_level</code>	Update the global var <code>debug_level</code> to the value written by userspace	0644
<code>llkdproc_show_pgoff</code>	retrieves (to userspace) the kernel's <code>PAGE_OFFSET</code> value	- no write callback -	0444
<code>llkdproc_show_drvctx</code>	retrieves (to userspace) the current values within the driver's 'context' structure <code>drv_ctx</code>	- no write callback -	0440
<code>llkdproc_config1</code> (also treated as <code>debug_level</code>)	retrieves (to userspace) the current value of the context variable <code>drvctx->config1</code>	update the driver context member <code>drvctx->config1</code> to the value written by userspace	0644

Table 13.1 : the four procfs files

The APIs and actual code to create the *proc_simple_intf* directory under */proc* and the four files mentioned above under it, follows shortly... do read on. (For lack of space, we don't actually show all the code; just the one wrt the 'debug level' get-and-set; it's not an issue, the remainder of the code is conceptually very similar).

Trying out the dynamic debug_level procfs control

Lets first check out the 'driver context' data structure that we shall use throughout this chapter (in fact, we first used it in *Chapter 11, Writing a simple Misc Character Device Driver*):

```
/* Borrowed from ch11; the 'driver context' data structure;
 * all relevant 'state info' reg the driver and (fictional) 'device'
 * is maintained here.
 */
struct drv_ctx {
    int tx, rx, err, myword, power;
    u32 config1; /* treated as equivalent to 'debug level' of our driver */
    u32 config2;
    u64 config3;
#define MAXBYTES    128
    char oursecret[MAXBYTES];
};
static struct drv_ctx *gdrvctx;
static int debug_level; /* 'off' (0) by default ... */
```

Above, we can also see that we have a global integer named `debug_level`; this will provide dynamic control on the debug verbosity for the 'project'. The debug level is assigned a range of [0-2], where:

- 0 implies *no debug messages* (the default)
- 1 is *medium debug* verbosity, and
- 2 implies *high debug* verbosity

The beauty of the whole schema and indeed the whole point here, is that we shall be able to query and set this `debug_level` variable from userspace via a procfs interface that we create! Thus allowing the end user (who, for security reasons, requires *root* access of course) to dynamically vary the debug level at runtime (a fairly common feature found in many products).

Before diving into the code-level detail, lets first try it out so that you can see what to expect:

1. Below, using our *lkm* convenience wrapper script, we build and `insmod(8)` the kernel module (here in the book's source tree: *ch13/proc_simple_intf*):

```
$ cd <booksrc>/ch13/proc_simple_intf
$ ../../lkm procfs_simple_intf          <-- builds the kernel
module
Version info:
[...]
```

```
[24826.234323] procfs_simple_intf:procfs_simple_intf_init():321:
proc dir (/proc/procfs_simple_intf) created
[24826.240592] procfs_simple_intf:procfs_simple_intf_init():333:
proc file 1 (/proc/procfs_simple_intf/llkdproc_debug_level) created
[24826.245072] procfs_simple_intf:procfs_simple_intf_init():348:
proc file 2 (/proc/procfs_simple_intf/llkdproc_show_pgoff) created
[24826.248628] procfs_simple_intf:alloc_init_drvctx():218:
allocated and init the driver context structure
[24826.251784] procfs_simple_intf:procfs_simple_intf_init():368:
proc file 3 (/proc/procfs_simple_intf/llkdproc_show_drvctx) created
[24826.255145] procfs_simple_intf:procfs_simple_intf_init():378:
proc file 4 (/proc/procfs_simple_intf/llkdproc_config1) created
[24826.259203] procfs_simple_intf initialized
$
```

Above, we built and inserted the kernel module; the `dmesg(1)` displays the kernel *printk*'s showing that (out of the four) one of the `procfs` files we created is the one pertaining to the dynamic debug facility (highlighted in bold font above; as these are pseudo files, the file size will show as 0 bytes).

2. Now, lets test it by querying the current value of `debug_level`:

```
$ cat /proc/procfs_simple_intf/llkdproc_debug_level
debug_level:0
$
```

3. Great, it's zero - the default - as expected; now lets change the debug level to 2:

```
$ sudo sh -c "echo 2 > /proc/procfs_simple_intf/llkdproc_debug_level"
$ cat /proc/procfs_simple_intf/llkdproc_debug_level
debug_level:2
$
```

Notice how we had to issue the `echo` as *root*; once done, the debug level has indeed changed (to the value 2)! Attempting to set the value out of range is caught as well (and the `debug_level` variable's value is reset to it's last valid value); we show this below:

```
$ sudo sh -c "echo 5 > /proc/procfs_simple_intf/llkdproc_debug_level"
sh: echo: I/O error
$ dmesg
[...]
[ 6756.415727] procfs_simple_intf: trying to set invalid value for
debug_level [allowed range: 0-2]; resetting to previous (2)
```

Right, it worked as expected. But, the question of course is: how did all this work at the level of the code? Read on to find out!

Dynamically controlling the debug_level via procfs

Here, we answer the above stated question - *how is it done in code?* It's quite straightforward, really...

1. First off, within the *init* code of the kernel module, we create our procfs directory naming it the name of our kernel module:

```
static struct proc_dir_entry *gprocdir;
[...]
```

```
gprocdir = proc_mkdir(OURMODNAME, NULL);
```

2. Again, within the *init* code of the kernel module, we create the procfs file that controls the project's 'debug level':

```
// code location: ch13/procfs_simple_intf/procfs_simple_intf.c

#define PROC_FILE1          "llkdpoc_debug_level"
#define PROC_FILE1_PERMS    0644
[...]
```

```
static int __init procfs_simple_intf_init(void)
{
    int stat = 0;
    [...]
    /* 1. Create the PROC_FILE1 proc entry under the parent dir OURMODNAME;
     * this will serve as the 'dynamically view/modify debug_level'
     * (pseudo) file */
    if (!proc_create(PROC_FILE1, PROC_FILE1_PERMS, gprocdir,
                    &fops_rdwr_dbg_level)) {
        [...]
        MSG("proc file 1 (/proc/%s/%s) created\n", OURMODNAME, PROC_FILE1);
        [...]
    }
}
```

Above, we use the *proc_create* API to create the procfs file and 'link' it with the supplied *file_operations* structure.

3. The 'fops' structure (technically, *struct file_operations*) is the key data structure here of course. As we learned in Ch 11, it's where we assign *functionality* to the various file operations on the device, or, as in this case, the procfs file; here's the code initializing our 'fops':

```
static const struct file_operations fops_rdwr_dbg_level = {
    .owner = THIS_MODULE,
    .open = myproc_open_dbg_level,
    .read = seq_read,
    .write = myproc_write_debug_level,
    .llseek = seq_lseek,
```

```
.release = single_release,
};
```

4. The *open* method of the 'fops' points to a function we define:

```
static int myproc_open_dbg_level(struct inode *inode, struct file
*file)
{
    return single_open(file, proc_show_debug_level, NULL);
}
```

Using the kernel's *single_open()* API, we register the fact that, whenever this file is read - which ultimately is via the *read(2)* system call from userspace - the proc filesystem will 'call back' our routine *proc_show_debug_level()* (the second parameter to *single_open()*).



We don't bother with the internal implementation of the *single_open()* here; if you're curious you can always look it up here:

fs/seq_file.c:single_open().

So, in summary, to register a 'read' method with the procs:

- initialize the *fops.open* pointer to a function *foo()*
- in function *foo()* call *single_open()* providing the read callback function as the second parameter



There's some history here; without getting too deep into it, suffice it to say that the older working of procs had issues. Notably, one couldn't transfer more than a single page of data (with read or write) without manually iterating over the content. The *sequence iterator* functionality introduced around 2.6.12 fixed these issues. Nowadays, using the *single_open()* and its ilk (the *seq_read*, *seq_lseek*, *seq_release* builtin kernel functions) is the simpler and correct approach to using procs.

5. To continue: what about when userspace *writes* (via the *write(2)* system call of course) into a proc file? Simple: above, you can see that we have registered the *fops_rdwr_dbg_level.write* method as the function *myproc_write_debug_level()*, implying that this function will be *called back* whenever this (pseudo) file is written to (it's explained in point 6 following the *read* callback below).

The code of the *read* callback function (we registered via the *single_open* above) follows below:

```

/* Our proc file 1: displays the current value of debug_level */
static int proc_show_debug_level(struct seq_file *seq, void *v)
{
    if (mutex_lock_interruptible(&mtx))
        return -ERESTARTSYS;
    seq_printf(seq, "debug_level:%d\n", debug_level);
    mutex_unlock(&mtx);
    return 0;
}

```

The `seq_printf()` is conceptually similar to the familiar `sprintf` API. It correctly prints - to the `seq_file` object - the data supplied to it. When we say 'prints' here, it really means that it effectively passes the data buffer to the userspace process or thread that issued the read system call that got us here in the first place, in effect *transferring the data to userspace*.



Oh yes, what's with the `mutex_{un}lock()` APIs above? They are for something critical - *locking*. We shall have to defer the detailed discussion on locking to Ch 16 and Ch 17; for now, just understand that these are required synchronization primitives.

6. The *write* callback function, we registered via the `fops_rdwr_dbg_level.write` above follows below:

```

#define DEBUG_LEVEL_MIN      0
#define DEBUG_LEVEL_MAX      2
[...]
/* proc file 1 : modify the driver's debug_level global variable as
per what userspace writes */
static ssize_t myproc_write_debug_level(struct file *filp,
                                         const char __user *ubuf, size_t count, loff_t *off)
{
    char buf[12];
    int ret = count, prev_dbglevel;
    [...]
    prev_dbglevel = debug_level;
    < ... validity checks ... >
    /* Get the usermode buffer content into the kernel (into 'buf')
    */
    if (copy_from_user(buf, ubuf, count)) {
        ret = -EFAULT;
        goto out;
    }
    [...]
    ret = kstrtoint(buf, 0, &debug_level); /* update it! */
    if (ret)
        goto out;
}

```

```

    if (debug_level < DEBUG_LEVEL_MIN || debug_level >
        DEBUG_LEVEL_MAX) {
        [...]
        debug_level = prev_dbglevel;
        ret = -EFAULT; goto out;
    }
    /* just for fun, lets say that our drv ctx 'config1'
       represents the debug level */
    gdrvctx->config1 = debug_level;
    ret = count;
out:
    mutex_unlock(&mtx);
    return ret;
}

```

In our write 'method' implementation above (notice how similar it is in structure to a character device driver's write method), we perform some validity checking, copy in the data the userspace process wrote to us (recall how in our example above we used the *echo* command to write to the procfs file) via the usual `copy_from_user()` function. We then use the kernel builtin `kstrtoint()` API (there are several in a similar vein) to convert the string buffer to an integer, storing the result into our global variable `debug_level`! Again, we validate it, and if all's well, we also set (just as an example) our driver context's `config1` member to the same value and then return success.

7. The remainder of the kernel module code is very similar - we setup the functionality for the remaining three procfs files. I leave it to you to browse through the code in detail and try it out.
8. One more quick demo: lets set the `debug_level` to 1 and then dump the driver context structure (via the third procfs file we created):

```

$ cat /proc/procfs_simple_intf/llkproc_debug_level
debug_level:0
$ sudo sh -c "echo 1 >
/proc/procfs_simple_intf/llkproc_debug_level"

```

9. Okay, the `debug_level` variable will now have the value 1; now, lets dump the driver context structure:

```

$ cat /proc/procfs_simple_intf/llkproc_show_drvctx
cat: /proc/procfs_simple_intf/llkproc_show_drvctx: Permission
denied
$ sudo cat /proc/procfs_simple_intf/llkproc_show_drvctx
prodname:procfs_simple_intf
tx:0,rx:0,err:0,myword:0,power:1
config1:0x1,config2:0x48524a5f,config3:0x424c0a52

```



```

oursecret:AhA xxx
$

```

We need *root* access to do so. Once done, we clearly see all the members of our `drv_ctx` data structure. Not only that, we verify that the member `config1` highlighted in bold font above, now has the value 1 reflecting the 'debug level' as designed.

Also, notice how the output is deliberately generated to userspace in a highly parse-able format, almost JSON-like. Of course, as a small exercise, you could arrange to do precisely that!

So, good; you have now seen the detailed steps on how exactly to create a `procfs` directory, a file within it, and, most importantly, how to create and use the read and write callback functions such that when a usermode process reads or writes your `proc` file, you can respond appropriately from deep within the kernel. As mentioned earlier, due to lack of space, we do not describe the code driving the remaining three `procfs` files we create and use; in any case, no worries, it's very similar conceptually to what we have just covered. We do expect you to read through and try it out!

A few misc `procfs` APIs

Lets conclude this section with a few remaining miscellaneous `procfs` APIs. You can create a symbolic or soft link within `/proc` by using the `proc_symlink()` function:

Next, the `proc_create_single_data()` API can be very useful; it's used as a 'shortcut' where one requires just a 'read' method attached to a `procfs` file:

```

struct proc_dir_entry *proc_create_single_data(const char *name, umode_t
mode, struct proc_dir_entry *parent, int (*show)(struct seq_file *, void
*), void *data);

```

Using this API thus eliminates the need for a separate 'fops' data structure. Indeed, we use precisely this function to create and work with our second `procfs` file - the `llkdpoc_show_pgoff` one:

```

... proc_create_single_data(PROC_FILE2, PROC_FILE2_PERMS, gprocdir,
proc_show_pgoff, 0) ...

```

When read from userspace, the kernel VFS and `proc` layer will invoke the registered method - the `proc_show_pgoff()` function of ours, within which we trivially invoke the `seq_printf()` to send the value of `PAGE_OFFSET` to userspace:

```

seq_printf(seq, "%s:PAGE_OFFSET:0x%lx\n", OURMODNAME, PAGE_OFFSET);

```

Further, and FYI regarding the `proc_create_single_data` API:

- You can make use of the fifth parameter to `proc_create_single_data()` to pass along any data item to the read callback (retrieved there as the `seq_file` member called `private`, very similar to how we used `filp->private_data` in the previous chapter)
- Several typically older drivers within the kernel mainline do make use of this function to create their procfs interfaces. Among them are the RTC driver (which sets up an entry here: `/proc/driver/rtc`). The SCSI *megaraid* driver (`drivers/scsi/megaraid.c`) uses this routine no fewer than 10 times to set up its proc interfaces (when a config option is enabled; it is by default).

Careful though! I find that on an Ubuntu 18.04 LTS system running the distro (default) kernel, this API - `proc_create_single_data()` - isn't even available, thus the build fails. On our custom 'vanilla' 5.4 LTS kernel it works just fine.



In addition, there is some documentation on the procfs API set here though these tend to be for internal usage and not for modules: <https://www.kernel.org/doc/html/latest/filesystems/api-summary.html#the-proc-file-system>.

So, as mentioned before, with the procfs APIs it's a case of YMMV (Your Mileage May Vary)! Carefully test your code before release.

This completes our coverage on using *procfs* as a useful communication interface. Lets move on now, to using a perhaps more appropriate one for drivers - the *sysfs* interface.

Interfacing via the sys filesystem (sysfs)

A critical feature of the 2.6 Linux release was the advent of what is called the modern *device model*. Essentially, a series of complex tree-like hierarchical data structures model all devices present on the system. Actually, it goes well beyond this; the sysfs tree encompasses (among other things):

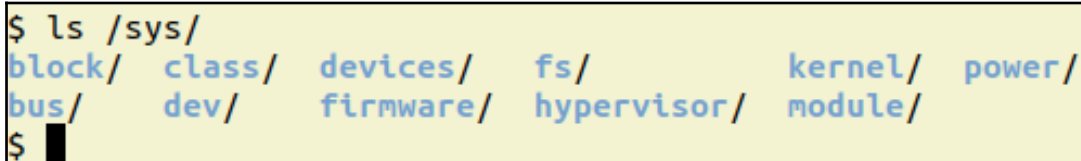
- every bus present on the system (it might be a virtual or pseudo bus as well)
- every device present on every bus
- every device driver bound to a device on a bus

Thus, it's not just peripheral devices but also the underlying system buses, the devices on

each bus and the device driver bound or that will bind to a device, that is runtime created and maintained by the device model. The inner workings of this model is quite invisible to you, as a typical driver author; you don't really have to worry about it. On system boot, and whenever a new device becomes visible, the *driver core* (part of the builtin kernel machinery) generates the required (virtual) files under the sysfs tree. (Conversely, when a device is removed or detached, its entry disappears from the tree).

Recall, though, from the section above on *Interfacing with the proc filesystem*, we learned that using *procfs* for a device driver's interfacing purposes is not really the right approach, at least for code that desires to move upstream. So what is the right approach? Ah, *creating sysfs (pseudo) files is considered the 'correct way' for device drivers to interface with userspace*.

So, now we see it! *sysfs* is a virtual filesystem typically mounted on `/sys`. In effect, *sysfs*, very similarly to *procfs*, is a kernel-exported tree of information (device and other) to userspace. You can think of *sysfs* as having different *viewports* into the modern device model. Via *sysfs*, you can view the system in several different ways or via different 'viewports'; for example, you can view the system via the various buses it supports (the *bus* view - *pci*, *usb*, *platform*, *i2c*, *spi*, among several others), via various 'classes' of devices (the *class* view), via the *devices* themselves, via the *block* devices viewport, and so on. A quick screenshot showing the content of `/sys` on our Ubuntu 18.04 LTS VM shows this to be the case:



```
$ ls /sys/
block/  class/  devices/  fs/      kernel/  power/
bus/    dev/    firmware/  hypervisor/  module/
$
```

Fig 13. : Screenshot showing the content of *sysfs* (*/sys*) on an x86_64 Ubuntu system

As can be seen, with *sysfs*, there are several other viewports via which you can look into the system as well. Of course, in this section, we wish to understand how to interface a device driver to userspace via *sysfs*; to write the code to create our driver (pseudo) files under *sysfs* and register the read/write callbacks from them. We begin by first looking at the basic *sysfs* APIs.

Creating a *sysfs* (pseudo) file in code

One way to to create a pseudo (or virtual) file under *sysfs* is via the `device_create_file()` API. Its signature is:

```
drivers/base/core.c:int device_create_file(struct device *dev,  
                                         const struct device_attribute *attr);
```

Lets consider it's two parameters one by one; first, is a pointer to `struct device`. The second parameter is a pointer to a device attribute structure; we shall explain and work upon it a bit later (in the section *Setting up the device attributes and creating the sysfs file*). For now, lets focus on the first parameter only - the device structure. It seems quite intuitive - a device is represented by a metadata structure called `device` (it is of course part of the driver core; you can look up it's full definition in the `include/linux/device.h` header).

A key point here is this: when you write (or work upon) a 'real' device driver, chances are high that a generic *device structure* will exist or come into being. This often happens upon *registration* of the device; an underlying device structure is usually made available as a member of a specialized structure for that device. For example, all structures such as `platform_device`, `pci_device`, `net_device`, `usb_device`, `i2c_client`, `serial_port` and so on, have a `struct device` member embedded within them. Thus, you can use that device structure pointer as a parameter to the API above for the purpose of creating files under `sysfs`. Rest assured, we shall soon see this being done in code! Hence, lets get going with getting ourselves a device structure by creating a simple 'platform device'. You should read about how to do this in the next section!

Creating a simple platform device

Clearly, in order to create a (pseudo) file under `sysfs`, we somehow require, as the first parameter to `device_create_file()`, a pointer to a `struct device`. However, for our demo *sysfs* driver here and now, we don't actually have any real device, and therefore no `struct device`, to work upon!

So, can't we create an *artificial* or *pseudo device* and simply use it? Yes, but how, and more crucially, why exactly should we have to do this? It's critical to understand that the modern *Linux Device Model* (the *LDM*) is built on three key components: **there must exist an underlying bus, upon which live devices; devices are 'bound to' and driven by device drivers.**

All of these must be registered to the driver core. Now, don't worry about the buses and the bus drivers that drive them, they will be registered and handled internally by the kernel's driver core subsystem. For the case where there is no real *device*, however, we will have to create a pseudo one in order to work with the model. Again, among several ways to do such things, we shall choose this method: we shall create a **platform device**. This device will 'live' on a pseudo bus (that is, it exists only in software), the *platform bus*.

Platform devices

A quick but important aside: *platform devices* are often used to represent the variety of devices on an *SoC* (*System on Chip*) within an embedded board. The SoC is typically a very sophisticated chip that integrates various components into its silicon - besides the processing units (CPUs/GPUs) it might house several peripherals too: among others, ethernet MAC, USB, multimedia, serial UART, clock, I2C, SPI, flash chip controllers, etc. A reason we require these components to be enumerated as a platform device is that there is no physical bus within the SoC; thus, the platform bus is used.



Traditionally, the code to instantiate these SoC platform devices was kept in a 'board' file (or files) within the kernel source; it getting overloaded has had it move outside the pure kernel source into a useful hardware description format called the *Device Tree* (within DTS source files that are themselves with the kernel source tree).

On our Ubuntu 18.04 LTS guest VM, we peek at the platform devices under sysfs:

```
$ ls /sys/devices/platform/
alarmtimer 'Fixed MDIO bus.0' intel_pmc_core.0 platform-
framebuffer.0 reg-dummy serial8250
eisa.0 i8042 pcspkr power rtc_cmos uevent
$
```



The *Bootlin* website (earlier named *Free Electrons*), offers superb materials on embedded Linux, drivers, etc. This link on their site leads to excellent material on the LDM (Linux Device Model).

Back to the driver: we bring into existence our (artificial) platform device by registering it to the (already existing) platform bus driver via the `platform_device_register_simple()` API. The moment we do so, the driver core will *generate* the required sysfs directories and a few boilerplate sysfs entries (or files). Below, in the *init* code of our sysfs demo driver, we setup a (simplest possible) *platform device* by registering it to the driver core:

```
// code location: ch13/sysfs_simple_intf/sysfs_simple_intf.c
include <linux/platform_device.h>
static struct platform_device *sysfs_demo_platdev;
[...]
#define PLAT_NAME    "llkd_sysfs_simple_intf_device"
sysfs_demo_platdev =
    platform_device_register_simple(PLAT_NAME, -1, NULL, 0);
[...]
```

The `platform_device_register_simple()` API returns a pointer to `struct platform_device`. One of this structure's members is `struct device dev`; so there, we now have what we've been after: a *device structure*. Also, it's key to note that when this registration API above runs, the effect is visible within *sysfs*. You can easily see (check it out below) the new platform device plus a few boilerplate *sysfs* objects created by the driver core (made visible to us via *sysfs*); let's build and *insmod* our kernel module to see this:

```
$ cd <...>/ch13/sysfs_simple_intf
$ make && sudo insmod ./sysfs_simple_intf.ko
[...]
```

```
$ ls -l /sys/devices/platform/llkd_sysfs_simple_intf_device/
total 0
-rw-r--r-- 1 root root 4096 Feb 4 16:31 driver_override
-r--r--r-- 1 root root 4096 Feb 4 16:31 modalias
drwxr-xr-x 2 root root 0 Feb 4 16:31 power/
lrwxrwxrwx 1 root root 0 Feb 4 16:31 subsystem -> ../../../../bus/platform/
-rw-r--r-- 1 root root 4096 Feb 4 16:31 uevent
$
```

FYI, several ways will exist to create a `struct device`; the generic way is to setup and issue the `device_create()` API. An alternate means to create a *sysfs* file, bypassing the need for a device structure, is to create a 'kobject' and invoke the `sysfs_create_file()` API. (We provide, in the *Further Reading* section, links to tutorials that uses both these approaches). Here, we prefer to use a 'platform device' as it's the closer approach to writing a (platform) driver.

Yet another valid approach: as shown in Ch 12, we built a simple character driver conforming to the kernel's *misc* framework. Here, we instantiated a `struct miscdevice`; once registered (via the `misc_register()` API), this structure will contain a member called `struct device *this_device`; thus allowing us to use it as a valid device pointer! We could simply extend our earlier 'misc' device driver and use it here. However, in order to learn a bit about platform drivers, we've chosen that approach. (We leave the approach of extending our earlier 'misc' device driver to use *sysfs* APIs and create/use *sysfs* files as an exercise to you).

Back to our driver, conversely to the *init* code, in the *cleanup* code, we must of course unregister our platform device:

```
platform_device_unregister(sysfs_demo_platdev);
```

Lets now tie all this knowledge together and actually generate the *sysfs* file along with it's read and write callback functions!

Tying it all together - setting up the device attributes and creating the sysfs file

As mentioned earlier, the `device_create_file()` API is the one we're using to create our sysfs file:

```
int device_create_file(struct device *dev, const struct device_attribute
*attr);
```

In the previous section we just saw how to obtain for ourselves a device structure (the first parameter to the API above). Now, let's figure out how to initialize and use the second parameter, the `device_attribute` structure. The structure itself is defined as follows:

```
include/linux/device.h

struct device_attribute {
    struct attribute attr;
    ssize_t (*show)(struct device *dev, struct device_attribute *attr,
                    char *buf);
    ssize_t (*store)(struct device *dev, struct device_attribute *attr,
                    const char *buf, size_t count);
};
```

The first member `attr`, essentially consists of the *name* of the sysfs file and its *mode* (permission bitmask). The other two members are function pointers ('virtual functions' just as in the file operations or 'fops' structure):

- `show`: represents the *read callback* function
- `store`: represents the *write callback* function

Our job is to initialize the structure, thus setting up the sysfs file. While you can always manually initialize it, there's an easier approach: the kernel provides (several) macros, to initialize the `struct device_attribute`; among them is the `DEVICE_ATTR()` macro:

include/linux/device.h:

```
define DEVICE_ATTR(_name, _mode, _show, _store) \
    struct device_attribute dev_attr_##_name = __ATTR(_name, _mode, _show,
    _store)
```

Notice the stringification performed via the `dev_attr_##_name`, ensuring that the structure's name is suffixed with the name passed as the first parameter to `DEVICE_ATTR`. Further, the actual 'worker' macro named `__ATTR()` (see below) actually instantiates a `device_attribute` structure in code at preprocessing time, with (via stringification) the

name of the structure becoming `dev_attr_<name>`:

include/linux/sysfs.h:

```
#define __ATTR(_name, _mode, _show, _store) { \
    .attr = { .name = __stringify(_name), \
    .mode = VERIFY_OCTAL_PERMISSIONS(_mode) }, \
    .show = _show, \
    .store = _store, \
}
```

Not only that, the kernel further defines additional simple wrapper macros over the above macros, to specify the *mode* (permissions for the sysfs file), thus making it even simpler for the driver author; among them are: `DEVICE_ATTR_RW(_name)`, `DEVICE_ATTR_RO(_name)`, `DEVICE_ATTR_WO(_name)` and so on; for example:

```
#define DEVICE_ATTR_RW(_name) \
    struct device_attribute dev_attr_##_name = __ATTR_RW(_name)
#define __ATTR_RW(_name) __ATTR(_name, 0644, _name##_show, _name##_store)
```

... using which, clearly, we can create a RW (read-write), read-only (RO) or write-only (WO) sysfs file. In our code, we wish to setup a sysfs 'file' that can be read and written to.

Internally, this is a 'hook' or callback for us to query or set a `debug_level` global variable just as we did in the sample kernel module on `procfs` earlier!

Now that we have sufficient background, let's delve into the code.

The code implementing our sysfs file creation and its callbacks

Here, we list relevant parts of the code for our simple *sysfs interfacing driver* and try things out, step by step:

1. Setup the device attribute structure (via the `DEVICE_ATTR_RW` macro; if unclear, see the immediately preceding section) and create our first sysfs (pseudo) file:

```
// code location: ch13/sysfs_simple_intf/sysfs_simple_intf.c

#define SYSFS_FILE1 llkdsysfs_debug_level
[... <we show the actual read/write callback functions below> ...]
static DEVICE_ATTR_RW(SYSFS_FILE1);

int __init sysfs_simple_intf_init(void)
{
    [...]
```



```
<< 0. The platform device is created via the
platform_device_register_simple() API; code already shown above ... >>

// 1. Create our first sysfile file : llkdsysfs_debug_level
/* The device_create_file() API creates a sysfs attribute file for
 * given device (1st parameter); the second parameter is the pointer
 * to it's struct device_attribute structure dev_attr<name> which was
 * instantiated by our DEV_ATTR{ _RW|RO } macros above ... */
static = device_create_file(&sysfs_demo_platdev->dev,
&dev_attr_SYSFS_FILE1);
[...]
```

From the definition of the macros shown above, we can infer that the line `static DEVICE_ATTR_RW(SYSFS_FILE1);` instantiates an initialized *device_attribute* structure with the name `llkdsysfs_debug_level` (as that's what the macro `SYSFS_FILE1` evaluates to) mode `0644`, the read callback name will be `llkdsysfs_debug_level_show()` and the write callback name will be `llkdsysfs_debug_level_store()`!

2. Here's the relevant code for the read and write callbacks (again, we don't show the entire code here); first, the *read callback*:

```
/* debug_level: sysfs entry point for the 'show' (read) callback */
static ssize_t llkdsysfs_debug_level_show(struct device *dev,
                                         struct device_attribute *attr,
                                         char *buf)
{
    int n;
    if (mutex_lock_interruptible(&mtx))
        return -ERESTARTSYS;
    MSG("In the 'show' method: name: %s, debug_level=%d\n",
        dev->kobj.name, debug_level);
    n = snprintf(buf, 25, "%d\n", debug_level);
    mutex_unlock(&mtx);
    return n;
}
```

How does this work? On reading our sysfs file, the above callback function is invoked. Within it, simply writing into the user-supplied buffer pointer `buf` (it's third parameter; we use the kernel `snprintf()` API to do so), has the effect of transferring the value provided (here the global `debug_level`) to userspace!

3. Lets try this much out: build and `insmod(8)` the kernel module (for convenience, we use our `lkm` wrapper script to do so):

```
$ ../../lkm sysfs_simple_intf      <-- build and insmod it
[...]
```

```
[83907.192247] sysfs_simple_intf:sysfs_simple_intf_init():237: sysfs file
[1]
(/sys/devices/platform/llkd_sysfs_simple_intf_device/llkdsysfs_debug_level)
created
[83907.197279] sysfs_simple_intf:sysfs_simple_intf_init():250: sysfs file
[2] (/sys/devices/platform/llkd_sysfs_simple_intf_device/llkdsysfs_pgoff)
created
[83907.201959] sysfs_simple_intf:sysfs_simple_intf_init():264: sysfs file
[3]
(/sys/devices/platform/llkd_sysfs_simple_intf_device/llkdsysfs_pressure)
created
[83907.205888] sysfs_simple_intf initialized
$
```

4. Now, lets list and read the sysfs file pertaining to the *debug level*:

```
$ ls -l
/sys/devices/platform/llkd_sysfs_simple_intf_device/llkdsysfs_debug_level
-rw-r--r-- 1 root root 4096 Feb  4 17:41
/sys/devices/platform/llkd_sysfs_simple_intf_device/llkdsysfs_debug_level
$ cat
/sys/devices/platform/llkd_sysfs_simple_intf_device/llkdsysfs_debug_level
0
```

Done; it reflects the fact that `debug_level` is currently 0.

5. Now, lets peek at the code of our *write callback* for the 'debug level' sysfs file:

```
#define DEBUG_LEVEL_MIN 0
#define DEBUG_LEVEL_MAX 2

static ssize_t llkdsysfs_debug_level_store(struct device *dev,
                                           struct device_attribute *attr,
                                           const char *buf, size_t count)
{
    int ret = (int)count, prev_dbglevel;
    if (mutex_lock_interruptible(&mtx))
        return -ERESTARTSYS;

    prev_dbglevel = debug_level;
    MSG("In the 'store' method:\ncount=%zu, buf=0x%p count=%zu\n",
        "Buffer contents: \".*s\\n\", count, buf, count, (int)count, buf);
    if (count == 0 || count > 12) {
        ret = -EINVAL;
        goto out;
    }

    ret = kstrtoint(buf, 0, &debug_level); /* update it! */
}
```

```

        < ... validity checks ... >
        ret = count;
out:
        mutex_unlock(&mtx);
        return ret;
}

```

Again, it should be clear that the `kstrtoint()` kernel API is used to convert the userspace `buf` string to an integer value, which we then validate. Also, the third parameter to `kstrtoint` is the integer to write to, thus updating it!

6. Now lets try updating the value of `debug_level` from it's sysfs file:

```

$ sudo sh -c "echo 2 >
/sys/devices/platform/llkd_sysfs_simple_intf_device/llkdsysfs_debug_level"
$ cat
/sys/devices/platform/llkd_sysfs_simple_intf_device/llkdsysfs_debug_level
2
$

```

Voila, it works.

7. Just as in the section on interfacing with *procfs* above, we have provided more code: another (read-only) sysfs interface to display the value of `PAGE_OFFSET`, plus a new one. Imagine that this driver's job is to retrieve a 'pressure' value (perhaps via an I2C-driven pressure sensor chip). Lets imagine we have done so, and stored this pressure value into an integer global variable named `gpressure`. To 'show' userspace the current pressure value, we do so via - the right approach! - a *sysfs file*. Here it is:



Internally, for the purpose of this demo, we randomly set the global variable `gpressure` to the value 25

```

$ cat
/sys/devices/platform/llkd_sysfs_simple_intf_device/llkdsysfs_pressure
25$

```

Why does the prompt appear immediately after the output? Because we just print the value as is - no newline, nothing. The code that displays the 'pressure' value is simple indeed:

```

/* show 'pressure' value: sysfs entry point for the 'show' (read) callback
*/
static ssize_t llkdsysfs_pressure_show(struct device *dev,
                                         struct device_attribute *attr, char *buf)

```

```

{
    int n;
    if (mutex_lock_interruptible(&mtx))
        return -ERESTARTSYS;
    MSG("In the 'show' method: pressure=%u\n", gpressure);
    n = snprintf(buf, 25, "%u", gpressure);
    mutex_unlock(&mtx);
    return n;
}
/* The DEVICE_ATTR{_RW|RO|WO}() macro instantiates a struct
device_attribute dev_attr_<name> here... */
static DEVICE_ATTR_RO(1lkdsysfs_pressure);

```

So, great, you've now learned how to interface with userspace via *sysfs*! As usual of course, I urge you to actually write code and try out these skills yourself; do see the *Questions* section at the end of the chapter and try out the (relevant) assignments yourself. Now, let's continue with *sysfs*, understanding an important 'rule' of its ABI.

The 'one value per sysfs file' rule

So far we have understood how to create and make use of *sysfs* for userspace-kernel interfacing purposes, but there is a key point that we have so far ignored. There is a 'rule' regarding the usage of *sysfs* files which says that one must take care to read or write exactly one value only! Think of this as the *one-value-per-file* rule.

So, with the example we used on a 'pressure' value above, we merely return the current value of the pressure, nothing more. Thus, *sysfs*, unlike the other interfacing technologies, is not quite suited to those cases where one might want to return arbitrary long-winded information packets (say, the contents of the driver context structure) to userspace; in other words, it's not very suited to pure 'debugging' purposes.



The kernel documents information and 'rules' regarding the usage of *sysfs* here: <https://www.kernel.org/doc/html/latest/admin-guide/sysfs-rules.html#rules-on-how-to-access-information-in-sysfs>.

In addition, there is documentation on the *sysfs* API set here: <https://www.kernel.org/doc/html/latest/filesystems/api-summary.html#the-filesystem-for-exporting-kernel-objects>.



FYI, the kernel typically provides several different means of creating *sysfs* objects; for example, with the *sysfs_create_files* API, one can create multiple *sysfs* files in one go: `int __must_check sysfs_create_files(struct kobject *kobj, const struct`



`attribute * const *attr);` here, you are expected to supply a pointer to a `kobject` and a pointer to a list of attribute structures.

This concludes our discussion of `sysfs` as an interfacing technology; in summary, `sysfs` is indeed considered the *right way* for driver authors to display and/or set a particular driver value to and from userspace. Due to the '*one value per sysfs file*' convention to be followed, `sysfs` is really not ideally suited to debugging information dispensation though; which neatly brings us to our next topic - *debugfs*!

Interfacing via the debug filesystem (debugfs)

Imagine for a moment, the quandary faced by you, a driver developer on Linux: you want to implement an easy yet elegant way to provide debug 'hooks' from your driver to userspace; for example, the user simply performing a *cat* on a (pseudo) file should result in your driver's 'debug callback' function being invoked. It will then proceed to dump some status information (perhaps a 'driver context' structure) to the usermode process, which will faithfully dump it to `stdout`.

Okay, no problem: in the days before the 2.6 release, we could happily use the *procfs* layer to interface our driver with userspace. Then from 2.6 Linux onward, the kernel community vetoed this approach. You were told to strictly stop using *procfs* and use the *sysfs* layer as the means to interface your driver with userspace. However, as we saw in the earlier section on `sysfs`, it has a pretty strict *one-value-per-file* rule. This quickly rules out all but the most trivial debug interfaces to userspace. We could use the *ioctl* approach (as we shall see), but it's quite a bit harder to do so.

So what do you do? Luckily, there is an elegant solution in place from around 2.6.12 Linux onward called *debugfs*. The 'debug filesystem' is very easy to use and quite explicit in communicating the fact that driver authors (anyone, in fact) can use it for whatever purpose they choose! There is no one-value-per-file rule; forget that, there are no rules.

Of course, just as with the other filesystem-based approaches we have dealt with, *procfs*, `sysfs` and now *debugfs*, the kernel community clearly claims that all these interfaces *are an ABI*, and thus, that their stability and lifespan is something that is *not* guaranteed. While that is the formal stance adopted, the reality is that these interfaces have become the de-facto ones in the real world; stripping them out without preamble one fine day wouldn't really serve anybody. The screenshot below shows the content of *debugfs* on our x86-64 Ubuntu 18.04.3 LTS guest (running the 'custom' 5.4.0 kernel we built back in Ch 3!):

```

root@llkd-vbox:~# uname -r
5.4.0-llkd01
root@llkd-vbox:~# mount |grep -w debugfs
debugfs on /sys/kernel/debug type debugfs (rw,relatime)
root@llkd-vbox:~# ls /sys/kernel/debug
acpi          dynamic_debug  opp           soundwire
bdi           error_injection pinctrl       split_huge_pages
block        extfrag       pmc_core     suspend_stats
cec          fault_around_bytes pm_qos       swiotlb
cleancache   frontswap     pwm          sync
clear_warn_once gpio          ras          tracing
clk          hid           regmap       usb
device_component iosf_sb       regulator    virtio-ports
devices_deferred kprobes      sched_debug  wakeup_sources
dma_buf      mce          sched_features x86
dri          memcg_slabinfo sleep_time    zswap
root@llkd-vbox:~# █

```

As with `procfs` and `sysfs`, `debugfs` being a *kernel* feature (it's a virtual filesystem after all!), the precise content within it is highly dependent on the kernel version and CPU architecture. As mentioned above, by looking at this screenshot, it should now be obvious that there are plenty of real-world 'users' of `debugfs`.

Checking for the presence of debugfs

First off, in order to make use of the powerful *debugfs* interface, it must be enabled within the kernel config. The relevant Kconfig macro is `CONFIG_DEBUG_FS`. Lets check whether it's enabled on our 5.4 custom kernel:



Here, we assume we have the `CONFIG_IKCONFIG` and the `CONFIG_IKCONFIG_PROC` options set to `y`, thus allowing us to use the `/proc/config.gz` pseudo-file to access the current kernel's configuration

```
$ zcat /proc/config.gz | grep -w CONFIG_DEBUG_FS
CONFIG_DEBUG_FS=y
```

Indeed it is; it's typically enabled by default in distributions.

Next, the default *mount point* of `debugfs` is `/sys/kernel/debug`. Thus we can see that it is internally dependant on the *sysfs* kernel feature being present and mounted, which of course, it is by default. Lets check where `debugfs` is mounted on our Ubuntu 18.04 x86-64

VM:

```
$ mount | grep -w debugfs  
debugfs on /sys/kernel/debug type debugfs (rw,relatime)
```

It is available and mounted at the expected location; good.



Of course, it's always best practice to never assume that this will always be the location where it's mounted though; in your script or usermode 'C' program, take the trouble to check and verify it. In fact, allow me to rephrase this: *it's always a good practice to never assume anything; making assumptions* is a really good source of bugs.

By the way, an interesting Linux feature is that filesystems can be mounted in different, even multiple locations; also, some folks prefer to create a symbolic link to `/sys/kernel/debug` under `/debug`; it's up to you really.

As usual, our intention here is to create our (pseudo) files under the debugfs umbrella, register and make use of the read/write callbacks from them, for the purpose of interfacing our driver with userspace. To do so, we need to understand the basic usage of the debugfs API; the section below will point you to documentation for precisely that purpose.

Looking up the debugfs API documentation

The kernel supplies succinct and superb documentation on using the debugfs API (courtesy Jonathan Corbet, LWN) here: <https://www.kernel.org/doc/Documentation/filesystems/debugfs.txt> (of course, you can also look it up directly within the kernel codebase).

I urge you to refer to this document for usage of the debugfs APIs, it's easy to read and understand; this way, we avoid unnecessarily repeating the same information here. In addition to the above document, the modern kernel documentation system (the 'Sphinx'-based one) also provides quite detailed debugfs API pages here: <https://www.kernel.org/doc/html/latest/filesystems/api-summary.html?highlight=debugfs#the-debugfs-filesystem>.



Note that all debugfs APIs are exported as GPL-only to kernel modules (thus necessitating the module being released under the 'GPL' license (can be dual licensed but one must be 'GPL')).

An interfacing example with debugfs

Debugfs, being deliberately designed with a 'no particular rules' mindset, makes it the ideal interface to use *for debug purposes*. Why? It allows you to construct any arbitrary byte stream and send it off to userspace, including a binary 'blob' with the `debugfs_create_blob()` API.

Our previous example kernel modules with *procfs* and *sysfs* as the means to provide interfaces, constructed and used 3 to 4 (pseudo) files. For a quick demo with *debugfs*, we shall just stick to two 'files':

- one, lets call `llkd_dbgfs_show_drvctx`; as you will have no doubt guessed, when read, it will cause the current content of our (by now familiar) 'driver context' data structure to be dumped to the console; we shall ensure it's read-only by root
- two, lets call `llkd_dbgfs_debug_level`; this one's mode shall be read-write (by root only); when read, it will display the current value of the global variable `debug_level`; when an integer is written to it, we shall update the value of the global variable `debug_level` to the value passed.

Below, in the *init* code of our kernel module, we first create a directory under debugfs:

```
// code: ch13/debugfs_simple_intf/debugfs_simple_intf.c

static struct dentry *gparent;
[...]
static int debugfs_simple_intf_init(void)
{
    int stat = 0;
    struct dentry *file1, *file2;
    [...]
    gparent = debugfs_create_dir(OURMODNAME, NULL);
```

Now that we have a starting point - a directory - lets move onto creating the debugfs (pseudo) files under it.

Creating and using the first debugfs file



For readability and to save space, we don't show the error handling code sections here.

Next, just as in the example with `procfs`, we allocate and initialize an instance of our 'driver context' data structure (we haven't shown the code here as it's repetitive, please refer to the GitHub source).

Then, via the generic `debugfs_create_file()` API, we create a `debugfs` 'file', associating it with a `file_operations` structure (in effect, getting the read callback registered):

```
static const struct file_operations dbgfs_drvctx_fops = {
    .read = dbgfs_show_drvctx,
};
[...]
```

```
< ... init function ... >
/* Generic debugfs file + passing a pointer to a data structure as a
 * demo.. the 4th param is a generic void * ptr; it's contents will be
 * stored into the i_private field of the file's inode.
 */
#define DBGFS_FILE1 "llkd_dbgfs_show_drvctx"
file1 = debugfs_create_file(DBGFS_FILE1, 0440, gparent,
    (void *)gdrvctx, &dbgfs_drvctx_fops);
[...]
```

Clearly, the 'read' callback is our function `dbgfs_show_drvctx()`, shown below; as a reminder, this function gets auto-invoked by the `debugfs` layer whenever the `debugfs` file `llkd_dbgfs_show_drvctx` is read; here's the code of our `debugfs` read callback function:

```
static ssize_t dbgfs_show_drvctx(struct file *filp, char __user * ubuf,
    size_t count, loff_t * fpos)
{
    struct drv_ctx *data = (struct drv_ctx *)filp->f_inode->i_private;
    // retrieve the "data" from the inode
#define MAXUPASS 256 // careful- the kernel stack is small!
    char locbuf[MAXUPASS];

    if (mutex_lock_interruptible(&mtx))
        return -ERESTARTSYS;

    /* As an experiment, we set our 'config3' member of the drv ctx stucture
     * to the current 'jiffies' value (# of timer interrupts since boot);
     * so, every time we 'cat' this file, the 'config3' value should change!
     */
    data->config3 = jiffies;
    snprintf(locbuf, MAXUPASS - 1,
        "prodname:%s\n"
        "tx:%d,rx:%d,err:%d,myword:%d,power:%d\n"
        "config1:0x%x,config2:0x%x,config3:0x%llx (%llu)\n"
        "oursecret:%s\n",
        OURMODNAME,
```

```

        data->tx, data->rx, data->err, data->myword, data->power,
        data->config1, data->config2, data->config3, data->config3,
        data->oursecret);

    mutex_unlock(&mtx);
    return simple_read_from_buffer(ubuf, MAXUPASS, fpos, locbuf,
                                   strlen(locbuf));
}

```

Notice how we retrieve the 'data' pointer (our driver context structure) by dereferencing the debugfs files' inode's member named `i_private`



As mentioned in Ch 12, this - using the data pointer to dereference the driver context structure from the file's inode - is one of a number of similar, common techniques employed by drivers to avoid the use of globals. Here of course, `gdrvctx` is a global, so it's a moot point; we simply use it to demonstrate the typical use case.

Using the `snprintf()` API we populate a local buffer with the current content of our driver's 'context' structure, and then, via the `simple_read_from_buffer()` API, pass it up to the userspace app that issued the read which typically causes it to be displayed on the terminal/console window. This API is a wrapper over the `copy_to_user()`.

Why not give it a spin:

```

$ ../../lkm debugfs_simple_intf
[...]
[200221.725752] dbgfs_simple_intf: allocated and init the driver context
structure
[200221.728158] dbgfs_simple_intf: debugfs file 1
<debugfs_mountpt>/dbgfs_simple_intf/llkd_dbgfs_show_drvctx created
[200221.732167] dbgfs_simple_intf: debugfs file 2
<debugfs_mountpt>/dbgfs_simple_intf/llkd_dbgfs_debug_level created
[200221.735723] dbgfs_simple_intf initialized

```

As can be seen from the `dmesg` output above, the two debugfs files are created as expected; lets verify (careful; you can only even look into debugfs as `root`):

```

$ ls -l /sys/kernel/debug/dbgfs_simple_intf
ls: cannot access '/sys/kernel/debug/dbgfs_simple_intf': Permission denied
$ sudo ls -l /sys/kernel/debug/dbgfs_simple_intf
total 0
-rw-r--r-- 1 root root 0 Feb  7 15:58 llkd_dbgfs_debug_level
-r--r----- 1 root root 0 Feb  7 15:58 llkd_dbgfs_show_drvctx
$

```

The files are created, and have the correct permissions. Lets now read (as root user) from the `llkd_dbgfs_show_drvctx` file:

```
$ sudo cat /sys/kernel/debug/dbgfs_simple_intf/llkd_dbgfs_show_drvctx
prodname:dbgfs_simple_intf
tx:0,rx:0,err:0,myword:0,power:1
config1:0x0,config2:0x48524a5f,config3:0x102fbcbc2 (4345023426)
oursecret:AhA yyy
$
```

It works; performing the read again a few seconds later, notice how the value of `config3` has changed. Why? recall ,we set it to the `jiffies` value - the number of timer 'ticks' - interrupts - that have ocured since system boot!:

```
$ sudo cat /sys/kernel/debug/dbgfs_simple_intf/llkd_dbgfs_show_drvctx |
grep config3
config1:0x0,config2:0x48524a5f,config3:0x102fbe828 (4345030696)
$
```

Having created and used our first debugfs file, lets now move onto understanding the second debugfs file.

Creating and using the second debugfs file

Lets move onto the second debugfs file; above, we create it using an interesting shortcut helper debugfs API named `debugfs_create_u32()`. This API *automatically* sets up internal callbacks allowing one to read/write upon the specified unsigned 32-bit global in the driver. The main advantage with this 'helper' routine is that you need not explicitly provide a `file_operations` structure or even any callback routines. The debugfs layer 'understands' and internally sets things up such that reading or writing the numeric (global) variable will always just work! Below, see the code in the *init* codepath that creates and sets up our second debugfs file:

```
static int debug_level;    /* 'off' (0) by default ... */
[...]
/* 3. Create the debugfs file for the debug_level global; we use the
 * helper routine to make it simple! There is a downside: we have no
 * chance to perform a validity check on the value being written.. */
#define DBGFS_FILE2        "llkd_dbgfs_debug_level"
file2 = debugfs_create_u32(DBGFS_FILE2, 0644, gparent, &debug_level);
[...]
pr_debug("%s: debugfs file 2 <debugfs_mountpt>/%s/%s created\n",
        OURMODNAME, OURMODNAME, DBGFS_FILE2);
```

It's as simple as that! Now, reading this file will produce the current value of the global

`debug_level`; writing to it will set it to the value written; lets do it:

```
$ sudo cat /sys/kernel/debug/dbgfs_simple_intf/llkd_dbgfs_debug_level
0
$ sudo sh -c "echo 5 >
/sys/kernel/debug/dbgfs_simple_intf/llkd_dbgfs_debug_level"
$ sudo cat /sys/kernel/debug/dbgfs_simple_intf/llkd_dbgfs_debug_level
5
$
```

It works, yes; but, look carefully though, there is a downside to this 'shortcut' approach : as it's all done internally, there is no way for us to *validate* the value being written. Thus above, we wrote the value 5 to `debug_level`; it worked, but it's an invalid value (at least lets assume that's the case)! So how can this be corrected? Simple: do not use this helper method; instead, do it the 'usual' way via the generic `debugfs_create_file()` API (as we did for the first debugfs file above). The advantage here, is that as we setup explicit callback routines for read and write, by specifying them within an 'fops' structure, we have control over the value being written (I leave doing this to you, as an exercise). Like life, it's a trade-off; you win something, you lose something.

Helper debugfs APIs for working on numeric globals

Above, we saw how we can make use of the `debugfs_create_u32()` helper API to setup a debugfs file to read/write an unsigned 32-bit integer global. The fact is, the debugfs layer provides a bunch of similar 'helper' APIs, to implicitly read/write on numeric (integer) global variables within your module.

Below, see the helper routines for creating debugfs entries that can read/write different bit size unsigned integer (8, 16, 32 and 64-bit) globals. The last parameter is the key one - the address of the global integer within the kernel/module:

```
// include/linux/debugfs.h
struct dentry *debugfs_create_u8(const char *name, umode_t mode,
                                struct dentry *parent, u8 *value);
struct dentry *debugfs_create_u16(const char *name, umode_t mode,
                                  struct dentry *parent, u16 *value);
struct dentry *debugfs_create_u32(const char *name, umode_t mode,
                                  struct dentry *parent, u32 *value);
struct dentry *debugfs_create_u64(const char *name, umode_t mode,
                                  struct dentry *parent, u64 *value);
```

The above APIs work with decimal base; to make using *hexadecimal base* easy, we have the

following:

```
struct dentry *debugfs_create_x8(const char *name, umode_t mode,
                                struct dentry *parent, u8 *value);
struct dentry *debugfs_create_x16(const char *name, umode_t mode,
                                struct dentry *parent, u16 *value);
struct dentry *debugfs_create_x32(const char *name, umode_t mode,
                                struct dentry *parent, u32 *value);
struct dentry *debugfs_create_x64(const char *name, umode_t mode,
                                struct dentry *parent, u64 *value);
```



As an aside, the kernel also provides a helper API for those cases where the precise *size* of the variable varies; hence, using the `debugfs_create_size_t()` helper creates a debugfs file appropriate for a variable of size `size_t`

For drivers that merely need to peek at a numeric global, or update it without any worry about invalid values, these debugfs helper APIs are very useful and are indeed commonly used by several drivers in the mainline kernel (we show an example within the MMC driver below). To evade the "validity check" issue, often, we can arrange for the *userspace* application (or script) to perform the validity checking; in fact, this is typically the 'right way' to do it. The UNIX paradigm, has a saying: *provide mechanism, not policy*.

When working with globals that are of *boolean* type, debugfs provides this helper API:

```
struct dentry *debugfs_create_bool(const char *name, umode_t mode,
                                  struct dentry *parent, bool *value);
```

Reading from the 'file' will result in only 'Y' or 'N' (suffixed with a newline) being 'returned'; obviously, 'Y' if the current value of the fourth parameter `value` is non-zero, 'N' otherwise. When writing, you can write 'Y' or 'N' or 1 or 0; all other values will not be accepted.



Think about it: you can control your 'robot' device via your robot device driver by writing 1 to a boolean variable called, say, *power* to turn it on, 0 to turn it off! ... the possibilities are endless.

The kernel documentation on debugfs provides a few more miscellaneous APIs; I leave it to you to have a look. We've now covered the creation and usage of our demo debugfs pseudo files; how do you remove them when done? The next section covers just this.

Removing the debugfs pseudo file(s)

When the module is removed (via, say, `rmmod(8)`), we must delete our debugfs files. The older way was via the `debugfs_remove()` API, where each debugfs file had to be individually removed with it (painful, to say the least). The modern approach makes it really simple:

```
void debugfs_remove_recursive(struct dentry *dentry);
```

Pass the pointer to the overall 'parent' directory (the one we created first), and the entire branch is recursively removed; perfect.

Not deleting your debugfs files at this point, thus leaving them there on the filesystem in an orphaned state is asking for trouble! Just think on this, what will happen when someone (attempts to) read or write to any of them later? A kernel bug, an *Oops*, that's what.

Seeing a kernel bug - an Oops!

Lets make it happen - a kernel bug! Exciting, yes!?

Okay, to create a kernel bug here is simple: (as the previous section hints at) we just ensure that on removal (unload) of the kernel module, the API that cleans up (deletes) all the debugfs files, `debugfs_remove_recursive()`, is *not* invoked. Thus, even after module removal, our debugfs directory and files seem to be present! But: if you try and operate (read/write) upon any of them, they're in an *orphaned state* and, hence, upon trying to dereference it's metadata, the internal debugfs code paths will perform an invalid memory reference resulting in a (kernel-level) bug.

In kernel-space, a bug is a very serious thing indeed; in theory, it should never ever occur! It's occurring is called an *Oops*; as part of the handling, an internal kernel function is called which dumps useful diagnostic information via *printk* (of course) to the in-memory kernel log buffer as well as to the console device (on production systems it might also be directed elsewhere so that it can be retrieved and investigated at a later point in time; for example, via the kernel's *kdump* mechanism).

Here's where we introduce a module parameter to control whether we (quite deliberately) cause an *Oops* to occur or not:

```
/* Module parameters */
static int cause_an_oops;
module_param(cause_an_oops, int, 0644);
MODULE_PARM_DESC(cause_an_oops,
    "Setting this to 1 can cause a kernel bug, an Oops; if 1, we do NOT
    perform required cleanup! so, after removal, any op on the debugfs files
```

```
will cause an Oops! (default is 0, no bug)");
[...]
```

In the cleanup code path of our driver, we check: if the variable `cause_an_oops` is non-zero, we deliberately do *not* (recursively) delete our debugfs file(s), hence setting up the bug:

```
static void debugfs_simple_intf_cleanup(void)
{
    kfree(gdrvctx);
    if (!cause_an_oops)
        debugfs_remove_recursive(gparent);
    pr_info("%s removed\n", OURMODNAME);
}
```

When we 'normally' perform the `insmod(8)`, the scary module parameter `cause_an_oops` is of course 0 by default, thus ensuring that everything works well. But lets get adventurous! we build the kernel module, and when inserting it, pass the parameter setting it to 1: (notice that below, we're running as *root* on our x86_64 Ubuntu 18.04 LTS guest system on our custom 5.4.0-11kd01 kernel):

```
# id
uid=0(root) gid=0(root) groups=0(root)
# insmod ./debugfs_simple_intf.ko cause_an_oops=1
# cat /sys/kernel/debug/dbgfs_simple_intf/llkd_dbgfs_debug_level
0
# dmesg
[ 2061.048140] dbgfs_simple_intf: allocated and init the driver context
structure
[ 2061.050690] dbgfs_simple_intf: debugfs file 1
<debugfs_mountpt>/dbgfs_simple_intf/llkd_dbgfs_show_drvctx created
[ 2061.053638] dbgfs_simple_intf: debugfs file 2
<debugfs_mountpt>/dbgfs_simple_intf/llkd_dbgfs_debug_level created
[ 2061.057089] dbgfs_simple_intf initialized (fyi, our 'cause an Oops'
setting is currently On)
#
```

Okay, now lets remove the kernel module - internally, the code to cleanup (recursively delete) our debugfs file would not have run. Next, we actually trigger the kernel bug, the *Oops*, by attempting to read one of our debugfs files:

```
# rmmod debugfs_simple_intf
# cat /sys/kernel/debug/dbgfs_simple_intf/llkd_dbgfs_debug_level
Killed
```

The message *Killed* on the console is a clue; something has gone (dramatically) wrong. Viewing the kernel log confirms that we indeed got an *Oops*! The (partially cropped)

screenshot below shows this:

```
[ 2119.775724] dbgfs_simple_intf removed
[ 2124.945311] BUG: unable to handle page fault for address: ffffffff8054d480
[ 2124.948501] #PF: supervisor read access in kernel mode
[ 2124.951069] #PF: error_code(0x0000) - not-present page
[ 2124.953575] PGD 7080e067 P4D 7080e067 PUD 70810067 PMD 7af5e067 PTE 0
[ 2124.956332] Oops: 0000 [#1] SMP PTI
[ 2124.958473] CPU: 1 PID: 4673 Comm: cat Tainted: G          OE      5.4.0-11kd01 #2
[ 2124.961171] Hardware name: innotek GmbH VirtualBox/VirtualBox, BIOS VirtualBox 12/01/2006
[ 2124.963971] RIP: 0010:debugfs_u32_get+0x5/0x20
[ 2124.966355] Code: e5 5d 48 89 06 31 c0 c3 0f 1f 00 66 2e 0f 1f 84 00 00 00 00 0f 1f 44 00 00 55 31 c0 89 37 48 89 e5 5d c3 90
0f 1f 44 00 00 <8b> 07 55 48 89 e5 5d 48 89 06 31 c0 c3 0f 1f 40 00 66 2e 0f 1f 84
[ 2124.973702] RSP: 0018:ffffa239808cbe00 EFLAGS: 00010246
[ 2124.976101] RAX: ffffffffbaa0b490 RBX: 0000000000000000 RCX: fffffa239808cbee8
[ 2124.978880] RDX: ffff92db34814440 RSI: fffffa239808cbe10 RDI: ffffffff8054d480
[ 2124.981827] RBP: fffffa239808cbe48 R08: ffffffffbb48a380 R09: 0000000000000000
[ 2124.984674] R10: 0000000000000000 R11: 0000000000000000 R12: ffff92db3cda0250
[ 2124.987504] R13: fffffa239808cbee8 R14: ffff92db3cda0200 R15: 0000000000020000
[ 2124.990426] FS: 00007f0e123d3540(0000) GS:ffff92db3db00000(0000) knlGS:0000000000000000
[ 2124.993462] CS: 0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[ 2124.996008] CR2: ffffffff8054d480 CR3: 000000004ccb001 CR4: 0000000000606e0
[ 2124.998808] Call Trace:
[ 2125.000850] ? simple_attr_read+0x6b/0xf0
[ 2125.003305] debugfs_attr_read+0x49/0x70
[ 2125.005576] __vfs_read+0x1b/0x40
[ 2125.007776] vfs_read+0x8e/0x130
[ 2125.009799] ksys_read+0xa7/0xe0
[ 2125.011934] __x64_sys_read+0x1a/0x20
[ 2125.013896] do_syscall_64+0x57/0x190
[ 2125.015921] entry_SYSCALL_64_after_hwframe+0x44/0xa9
[ 2125.018103] RIP: 0033:0x7f0e11ee0081
[ 2125.020150] Code: fe ff ff 48 8d 3d 67 9c 0a 00 48 83 ec 08 e8 a6 4c 02 00 66 0f 1f 44 00 00 48 8d 05 81 08 2e 00 8b 00 85 c0 75
13 31 c0 0f 05 <48> 3d 00 f0 ff ff 77 57 f3 c3 0f 1f 44 00 00 41 54 55 49 89 d4 53
[ 2125.027032] RSP: 002b:00007ffceb55a5a8 EFLAGS: 00000246 ORIG_RAX: 0000000000000000
[ 2125.029474] RAX: ffffffff8054d480 RBX: 0000000000020000 RCX: 00007f0e11ee0081
[ 2125.032055] RDX: 0000000000020000 RSI: 00007f0e123b1000 RDI: 0000000000000003
[ 2125.034592] RBP: 0000000000020000 R08: 00000000ffffffff R09: 0000000000000000
[ 2125.037051] R10: 0000000000000022 R11: 0000000000000246 R12: 00007f0e123b1000
[ 2125.039547] R13: 0000000000000003 R14: 00007f0e123b100f R15: 0000000000020000
[ 2125.041867] Modules linked in: vboxsf(OE) vboxvideo(OE) vmwgfx drn_kms_helper syscopyarea sysfillrect snd_intel8x0 sysimgblt snd
```

Fig 13. : Partial screenshot of a kernel Oops, a kernel-level bug

As kernel debugging details go beyond the scope of this book, we do not delve into details here. Nevertheless, some of it at least is quite intuitive: look carefully, you can see in the BUG: statement the kernel virtual address (kva) whose lookup caused the bug, the Oops. The line

```
CPU: 1 PID: 4673 Comm: cat Tainted: G OE 5.4.0-11kd01 #2
```

shows the CPU (1) that the process context (*cat*) was running upon, the tainted flags and the kernel version. One of the really key pieces of output is this:

```
RIP: 0010:debugfs_u32_get+0x5/0x20
```

... telling you that the CPU instruction pointer (the register named RIP on the x86_64) was in the function `debugfs_u32_get()` at an offset of 0x5 bytes from the start of the machine code of the function (and it thinks that the length of the function is 0x20 bytes)!



Combining this information with powerful tools like `objdump(1)` and `addr2line(1)` can help to literally pinpoint the location of the bug in code!

The CPU registers are dumped, as well as - very useful! - the *Call Trace* or the *call stack* - the *content of the kernel-mode stack* of the process context (refer back to *Ch 6* for details on the kernel stack), showing you the code that led up to this point, the crash (read it bottom-up of course). Another quick tip: if a kernel function in the call trace output is preceded by a ? symbol, just ignore it (it's perhaps a previous 'blip' left behind).



Realistically, a kernel bug on a production system *must* cause the entire system to panic (halt); on non-production systems (like what we're running on), a kernel panic may or may not occur; here it doesn't. Nevertheless, a kernel bug must be treated with the highest level of severity, it's indeed a show-stopper and must be fixed. FYI, the `procfs` file `/proc/sys/kernel/panic_on_oops` is set to 0 by most distros but on production will typically be 1.

The moral here is clear: there is no auto cleanup performed by `debugfs`; we have to do it. Right, let's wrap up the discussion on `debugfs` by looking up some actual real-world usage within the kernel.

Debugfs - actual users

As mentioned previously, there are several 'real-world' users of the `debugfs` API; can we spot some of them? Well, here's one way: simply search under the kernel source tree's `drivers/` directory for files named `*debugfs*.c`; you might be surprised (I found 114 such files in the 5.4.0 kernel tree!); we show a few below:

```
$ cd ${LLKD_KSRC} ; find drivers/ -iname "*debugfs*.c"
drivers/block/drbd/drbd_debugfs.c
drivers/mmc/core/debugfs.c
drivers/platform/x86/intel_telemetry_debugfs.c
[...]
drivers/infiniband/hw/qib/qib_debugfs.c
drivers/infiniband/hw/hfi1/debugfs.c
[...]
drivers/media/usb/uvc/uvc_debugfs.c
drivers/acpi/debugfs.c
drivers/net/wireless/mediatek/mt76/debugfs.c
[...]
drivers/net/wireless/intel/iwlwifi/mvm/debugfs-vif.c
```

```
drivers/net/wimax/i2400m/debugfs.c
drivers/net/ethernet/broadcom/bnxt/bnxt_debugfs.c
drivers/net/ethernet/marvell/mvpp2/mvpp2_debugfs.c
drivers/net/ethernet/mellanox/mlx5/core/debugfs.c
[...]
drivers/misc/genwqe/card_debugfs.c
drivers/misc/mei/debugfs.c
drivers/misc/cxl/debugfs.c
[...]
drivers/usb/mtu3/mtu3_debugfs.c
drivers/sh/intc/virq-debugfs.c
drivers/soundwire/debugfs.c
[...]
drivers/crypto/ccree/cc_debugfs.c
```

Have a look at (some of) them; their code exposes *debugfs interfaces*; and, realize this, it's not always for mere debug purposes, many of the debugfs 'files' are for actual production usage! As just one example from the above list, the MMC driver has the following line of code that makes use of the debugfs 'helper' API for an x32 global:

```
drivers/mmc/core/debugfs.c:mmc_add_card_debugfs():
debugfs_create_x32("state", S_IRUSR, root, &card->state);
```

... which of course creates a debugfs file called `state`, which when read, displays the 'state' of the card.

Okay, this completes the coverage on how you can interface with userspace via the powerful debugfs framework. Our demo debugfs 'driver' created a debugfs directory and two debugfs pseudo files within it; we then learned how to setup and use both read and write callback handlers for them. The 'shortcut' APIs (like the `debugfs_create_u32()` and friends) are indeed powerful too. Not only that, we even managed to generate a kernel bug, an Oops! Now, let's now move on to communicating over a special type of socket, a *netlink socket*.

Interfacing via netlink sockets

Here, we'll learn to interface kernel and user spaces with a familiar and indeed ubiquitous network abstraction - *sockets*! Programmers familiar with network application programming can swear by its advantages.



Familiarity with the de-facto *network programming in C/C++ with socket APIs* helps here. Do see the *Further Reading* section for a couple of good tutorials on this topic.

Advantages using sockets

Among others, socket technology gives one several *advantages* (over other typical IPC mechanisms like pipes, SysV IPC / POSIX IPC mechanisms - message queues, shared memory, semaphores - etc); they include:

- bidirectional simultaneous data transfer (full duplex)
- lossless on the internet, with at least with some transport layer protocols, like TCP, and of course, on the localhost, which is the case here
- high speed data transfer especially on localhost!
- flow control semantics are always in effect
- asynchronous communication; messages can be queued, the sender does not have to wait for the receiver
- especially with respect to our topic: in other user<->kernel communication paths (like `procfs`, `sysfs`, `debugfs`, `ioctl`), the userspace app must initiate the transfer to kernel-space; with netlink sockets, *the kernel can initiate a transfer*
- also, with all the other mechanisms we have seen so far (`procfs`, `sysfs`, `debugfs`), the various interface files strewn all over the filesystem(s) can cause kernel namespace pollution; with netlink sockets (and incidentally, with the `ioctl`, that follows), this isn't the case as there are no 'files'.

These advantages can be helpful depending on the type of product you're working upon. Lets now get to particulars regarding the netlink socket.

Understanding what a netlink socket is

So, what is a netlink socket? We shall keep it simple - a *netlink socket* is a 'special' socket family that exists only on the Linux OS since the 2.2 version. Using it, one can setup **IPC** or **Inter Process Communication** between a usermode process (or thread) and a component within the kernel; in our case, a kernel module, typically a driver.

It is similar to a UNIX domain datagram socket in many ways; it's meant for communication on the *localhost only* and not across systems. While UNIX domain sockets use a pathname as their namespace, netlink sockets use a PID. Pedantically, this is a Port ID

and not a Process ID, although realistically, Process IDs are very often used as the namespace. The modern kernel core (besides drivers) uses netlink sockets in many cases - as one example, the `iproute2` networking utilities use it to configure wireless drivers; as another interesting example, the `udev` feature uses netlink sockets to effect communication between the kernel `udev` implementation and the userspace daemon process (`udev` or `systemd-udev`, for things like device discovery, device node provisioning, etc).

Below, we shall design and implement a simple user<->kernel messaging demonstration using netlink sockets. To do so, we shall have to write two programs (at a minimum) - one, the userspace application that issues socket-based system calls, and the other for the kernel-space component (here, a kernel module). We shall have the userspace process send a 'message' to the kernel module; the kernel module should receive it and print it (into the kernel log buffer). The kernel module will then reply to the userspace process, which is blocking on this very event.

So, without further ado, let's dive into writing code using netlink sockets; we shall begin with the userspace application. Read on!

Writing the userspace netlink socket application

A series of steps is required to get the *userspace* application running; follow along below:

1. The first thing to do is to get ourselves a *socket*. Traditionally, a socket is defined as an endpoint of communication; thus a pair of sockets forms a *connection*. We use the `socket(2)` system call to do so; its signature is:

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

Without getting into too much detail, here's what we do:

 - we specify the 'domain' as the special family `PF_NETLINK`, thus requesting a netlink socket
 - 'type' as `SOCK_RAW`, using a raw socket (effectively skipping the transport layer)
 - 'protocol' is the protocol to use; as we're using a raw socket, the protocol is left to be implemented either by us or by the kernel; having the kernel netlink code do it is the right approach; here, we use an unused protocol number 31
2. The next step is to bind the socket via the 'usual' `bind(2)` system call semantics; we first initialize a netlink source `socketaddr` structure for this purpose (where we specify the family as netlink, the PID value as the calling process' PID and for unicast only); the code below reveals the above two steps (for clarity, we don't display the error checking code below):

```
// ch13/netlink_simple_intf/userapp_netlink/netlink_userapp.c
#define NETLINK_MY_UNIT_PROTO      31
    // kernel netlink protocol # (registered by our kernel module)
#define NLSPACE 1024

[...]
```

/* 1. Get ourselves an endpoint - a netlink socket! */

```
sd = socket(PF_NETLINK, SOCK_RAW, NETLINK_MY_UNIT_PROTO);
printf("%s:PID %d: netlink socket created\n", argv[0], getpid());

/* 2. Setup the netlink source addr structure and bind it */
memset(&src_nl, 0, sizeof(src_nl));
src_nl.nl_family = AF_NETLINK;
/* Note carefully: nl_pid is NOT necessarily the PID of the sender
process; it's actually 'port id' and can be any unique number */
src_nl.nl_pid = getpid();
src_nl.nl_groups = 0x0; // no multicast
bind(sd, (struct sockaddr *)&src_nl, sizeof(src_nl))
```

3. Next, initialize a netlink 'destination address' structure; here, we set the PID member to 0, a special value indicating that the destination is the kernel:

```
/* 3. Setup the netlink destination addr structure */
memset(&dest_nl, 0, sizeof(dest_nl));
dest_nl.nl_family = AF_NETLINK;
dest_nl.nl_groups = 0x0; // no multicast
dest_nl.nl_pid = 0;      // destined for the kernel
```

4. Next, we allocate and initialize a netlink 'header' data structure; (among other things) it specifies the source PID and, importantly, the data 'payload' that we shall deliver to our kernel component (as can be seen below, we make use of helper macros like the NLMSG_DATA() to specify the correct data location within the netlink header structure):

```
/* 4. Allocate and setup the netlink header (including the payload) */
nlhdr = (struct nlmsghdr *)malloc(NLMSG_SPACE(NLSPACE));
memset(nlhdr, 0, NLMSG_SPACE(NLSPACE));
nlhdr->nlmsg_len = NLMSG_SPACE(NLSPACE);
nlhdr->nlmsg_pid = getpid();
/* Setup the payload to transmit */
strncpy(NLMSG_DATA(nlhdr), thedata, strlen(thedata)+1);
```

5. Next, an iovec structure is initialized to reference the netlink header, and a msghdr datat structure is initialized to point to the destination address and the (above-mentioned) iovec:

```
/* 5. Setup the iovec and ... */
```

```
memset(&iiov, 0, sizeof(struct iovec));
iov.iov_base = (void *)nlhdr;
iov.iov_len = nlhdr->nlmsg_len;
[...]
/* ... now setup the message header structure */
memset(&msg, 0, sizeof(struct msghdr));
msg.msg_name = (void *)&dest_nl; // dest addr
msg.msg_namelen = sizeof(dest_nl); // size of dest addr
msg.msg_iov = &iiov;
msg.msg_iovlen = 1; // # elements in msg_iov
```

6. Finally (!) the message is sent (transmitted) via the `sendmsg(2)` system call (which takes the socket descriptor and the above-mentioned `msghdr` structure as a parameter):

```
/* 6. Actually (finally!) send the message via sendmsg(2) */
nsent = sendmsg(sd, &msg, 0);
```

7. The kernel component - a kernel module that we shall discuss below - should now receive the message via *it's* netlink socket and display the message content; it will then politely reply. To grab the reply, our userspace app must now perform a blocking read on the socket:

```
/* 7. Block on incoming msg from the kernel-space netlink component */
printf("%s: now blocking on kernel netlink msg via recvmsg() ...\n",
argv[0]);
nrecv = recvmsg(sd, &msg, 0);
```

We employ the `recvmsg(2)` system call to do so, and when it gets unblocked, print out the message received.



Why so much abstraction and wrapping of data structures? Well, it's how things often evolve - the `msghdr` structure was created so that the `sendmsg(2)` API can use fewer parameters. But that implies the parameters have to go somewhere; they go deep inside `msghdr`, which points to the destination address and the `iovec`, whose (`iovec`'s) base points to the netlink header structure, which contains the payload! Whew.

As an experiment, what if we build and run the usermode netlink application prematurely - *without* the kernel-side code in place? It will fail of course... But how exactly? Well, use the empirical approach: trying it out via the venerable `strace(1)` utility, we can see that the `socket(2)` system call returns failure, the cause being "Protocol not supported":

```
$ strace -e trace=network ./netlink_userapp
socket(AF_NETLINK, SOCK_RAW, 0x1f /* NETLINK_??? */) = -1 EPROTONOSUPPORT
(Protocol not supported)
```

```
netlink_u: netlink socket creation failed: Protocol not supported
+++ exited with 1 +++
$
```

Quite right; there is no such 'protocol # 31' (31 = 0x1f, the protocol number we're using) in place *yet* within the kernel! We're yet to do this. So that's the userspace side of things. Lets now complete the puzzle and have it actually work! by seeing how the kernel component (module) is written.

Writing the kernel-space netlink socket code as a kernel module

The kernel provides the base infrastructure for netlink, including APIs and data structures; all required ones are exported and thus available to you as a module author. We use several of them; the steps to program our kernel netlink component - our kernel module - are outlined below:

1. Just as with the userspace app, the first thing to do is to get ourselves a netlink socket; the kernel API is the `netlink_kernel_create()`; it's signature is:

```
struct sock * netlink_kernel_create(struct net *, int , struct
netlink_kernel_cfg *);
```

The second parameter is the *protocol number (unit)* to use; we shall specify the same number (31) as we did for the userspace app. The third parameter is a pointer to an (optional) netlink configuration structure; here, we only set the input member to a function of ours nullifying the rest. This function is 'called back' when a userspace process (or thread) provides any input (that is transmits something) to the kernel netlink component. So, within our kernel module's 'init' routine:

```
//
ch13/netlink_simple_intf/kernelspace_netlink/netlink_simple_intf.c
#define OURMODNAME "netlink_simple_intf"
#define NETLINK_MY_UNIT_PROTO 31
// kernel netlink protocol # that we're registering
static struct sock *nlsock;
[...]
static struct netlink_kernel_cfg nl_kernel_cfg = {
    .input = netlink_recv_and_reply,
};
[...]
nlsock = netlink_kernel_create(&init_net, NETLINK_MY_UNIT_PROTO,
```

```
&nl_kernel_cfg);
```

2. As mentioned, when a userspace process (or thread) provides any input (that is, transmits something) to our kernel (netlink) module, the callback function is invoked. It's important to understand that it runs in process context and not any kind of interrupt context; we use our `convenient.h:PRINT_CTX()` macro to verify the same. Here, we simply display the received 'message' and then reply by sending a sample message to our userspace peer process.

The data payload transmitted from our userspace peer process can be retrieved from the socket buffer structure that is passed along to our callback function as a parameter from a netlink header structure within it. See how the data and sender PID are retrieved in the code snippet below:

```
static void netlink_rcv_and_reply(struct sk_buff *skb)
{
    struct nlmsghdr *nlh;
    struct sk_buff *skb_tx;
    char *reply = "Reply from kernel netlink";
    int pid, msgsz, stat;

    /* Find that this code runs in process context, the process
     * (or thread) being the one that issued the sendmsg(2) */
    PRINT_CTX();

    nlh = (struct nlmsghdr *)skb->data;
    pid = nlh->nlmsg_pid; /*pid of sending process */
    pr_info("%s: received from PID %d:\n"
           "\"%s\"\n", OURMODNAME, pid, (char *)NLMSG_DATA(nlh));
```



FYI, the *socket buffer* data structure - `struct sk_buff` - is considered the critical data structure within the Linux kernel's network protocol stack. It holds all metadata concerning the network 'packet', including dynamic pointers to it. It has to be quickly allocated and freed (especially when network code runs in interrupt contexts); this is indeed possible because it's on the kernel's slab (SLUB) cache (see *Ch 7* for details on the kernel slab allocator).

Here and now, all we need to understand is that we can retrieve the payload from the network packet by first dereferencing the `data` member of the socket buffer (`skb`) structure passed to our callback routine! Next, this `data` member is actually the pointer to the netlink message header structure setup by our userspace peer; we then dereference it to get the actual payload.

3. We would now like to 'reply' to our userspace peer process; doing so involves a few actions.

Firstly, we allocate a new netlink message with the `nlmsg_new()` API which is really a thin wrapper over `alloc_skb()`, then add a netlink message to the just allocated socket buffer via the `nlmsg_put()` API and then copy in the data (the 'payload') into the netlink header using an appropriate macro (`nlmsg_data()`):

```
//--- Lets be polite and reply
msgsz = strlen(reply);
skb_tx = nlmsg_new(msgsz, 0);
[...]
// Setup the payload
nlh = nlmsg_put(skb_tx, 0, 0, NLMSG_DONE, msgsz, 0);
NETLINK_CB(skb_tx).dst_group = 0; /* unicast only (cb is the
    * skb's control buffer), dest group 0 => unicast */
strncpy(nlmsg_data(nlh), reply, msgsz);
```

4. The actual 'sending' of the reply to our userspace peer process is achieved here via the `nlmsg_unicast()` API (which tells us that even multicasting netlink messages are possible; we don't use that here):

```
// Send it
stat = nlmsg_unicast(nlsock, skb_tx, pid);
```

5. That only leaves the cleanup (which is of course invoked when the kernel module is removed); the `netlink_kernel_release()` API is effectively the inverse of the `netlink_kernel_create()`; it 'cleans up' the netlink socket, shutting it down:

```
static void __exit netlink_simple_intf_exit(void)
{
    netlink_kernel_release(nlsock);
    pr_info("%s: removed\n", OURMODNAME);
}
```

Okay, now that we have written both the userspace app and the kernel module to interface via a netlink socket, lets actually try it out!

Trying out our netlink interfacing project

It's time to verify it all works as advertised; below, we show the steps in trying it out.

1. First, build and insert the kernel module into kernel memory:



Our lkm convenience script makes short work of that; the session below was carried out on our familiar x86_64 guest VM running Ubuntu 18.04 LTS and our custom 5.4.0 Linux kernel

```
$ cd <booksrc>/ch13/netlink_simple_intf/kernelspace_netlink
$ ../../../../lkm netlink_simple_intf
Version info:
Distro:      Ubuntu 18.04.4 LTS
Kernel: 5.4.0-11kd01
[...]
make || exit 1
[...] Building for: KREL=5.4.0-11kd01 ARCH=x86 CROSS_COMPILE=
EXTRA_CFLAGS= -DDEBUG
CC [M]
/home/11kd/booksrc/ch13/netlink_simple_intf/kernelspace_netlink/net
link_simple_intf.o
[...]
sudo insmod ./netlink_simple_intf.ko && lsmod|grep
netlink_simple_intf
-----
netlink_simple_intf    16384  0
[...]
[58155.082713] netlink_simple_intf: creating kernel netlink socket
[58155.084445] netlink_simple_intf: inserted
$
```

2. Okay, great, it's loaded up and ready. Next, we build and try out our userspace application:

```
$ cd ../userapp_netlink/
$ make netlink_userapp
[...]
```

The screenshot below reveals the action:

```

$ lsmod |grep netlink_simple_intf
netlink_simple_intf    16384  0
$
$ ../userapp_netlink/netlink_userapp
../userapp_netlink/netlink_userapp:PID 7813: netlink socket created
../userapp_netlink/netlink_userapp: bind done
../userapp_netlink/netlink_userapp: destination struct, netlink hdr, payload setup
../userapp_netlink/netlink_userapp: initialized iov structure (nl header folded in)
../userapp_netlink/netlink_userapp: initialized msghdr structure (iov folded in)
../userapp_netlink/netlink_userapp:sendmsg(): *** success, sent 1040 bytes all-inclusive
(see kernel log for dtl)
../userapp_netlink/netlink_userapp: now blocking on kernel netlink msg via recvmsg() ...
../userapp_netlink/netlink_userapp:recvmsg(): *** success, received 44 bytes:
msg from kernel netlink: "Reply from kernel netlink"
$
$ dmesg
[62818.385716] netlink_simple_intf: creating kernel netlink socket
[62818.389860] netlink_simple_intf: inserted
[62838.889120] netlink_recv_and_reply(): [000] netlink_userapp :7813 | ...0
[62838.900928] netlink_simple_intf: received from PID 7813:
"sample user data to send to kernel via netlink"
[62838.922712] netlink_simple_intf: reply sent
$ █

```

Fig 13. : Screenshot showing user->kernel communication via our sample netlink socket code

It works; the kernel netlink module receives and displays the message sent to it from the userspace process (PID 7813 above). The kernel module then replies with it's own message to it's userspace peer, which successfully receives and displays it (via a `printf()`). Do give it a spin yourself. When done, don't forget to remove the kernel module with a `sudo rmmod netlink_simple_intf`.



An aside: a 'connector' driver exists within the kernel. It's purpose is to ease the development of netlink-based communication, making it simpler for both the kernel and userspace developer to setup and use a netlink-based communication interface. We do not delve into it here; please refer it's documentation within the kernel. Also, sample code is provided within the kernel source tree as well (at `samples/connector`).

There, you have now learned how to interface between a usermode app and a kernel component via the powerful netlink socket mechanism. As mentioned earlier, it has several actual use cases within the kernel tree. Now, lets move on to covering one more user-kernel interfacing method - via the popular `ioctl(2)` system call.

Interfacing via the `ioctl` system call

The **ioctl**, firstly, is a system call; why the funny name *ioctl*? It's an abbreviation for **input-output control**. While the read and write system calls (among others) are used to effectively transfer *data* from and to a device (or file; remember the UNIX paradigm- *if it's not a process, it's a file!*), the *ioctl* system call is used to *issue commands* to the device (via it's driver of course). For example, changing a console device's terminal characteristics, writing a track of a disk when formatting it, sending a control command to a stepper motor, controlling a camera or audio device, and so on are all instances of *commands* being sent to a device.

Lets consider a fictitious example. We have a device and are developing a (character) driver for it. The device has various *registers*, small - typically 8, 16 or 32-bit pieces of hardware memory on the device - some of which are control registers. By appropriately performing I/O (reads and writes) on them, we in effect control the device (well, that's really the whole point, isn't it; the actual subject matter regarding working with hardware memory including device registers will be covered in *Ch 15*). So how will the driver author communicate or *interface* with a userspace program that wants to perform various control operations on this device? We often architect the userspace 'C' (or C++) program to open the device typically by performing an `open(2)` on it's *device file*, and subsequently issue the read and write system calls.

But, as we just mentioned above - the `read(2)` and `write(2)` system call APIs are appropriate when *transferring data* while here, instead, we intend to perform **control operations**. So, we need another system call to do so... Do we then require to create and encode a new system call (or calls)? No, it's much simpler that that: we *multiplex via the ioctl system call*, leveraging it to perform any required control operations upon our device. How? Ah, recall from the previous chapter the all-important `file_operations` ('fops') data structure; we now initialize another member, the `.ioctl` one, to our `ioctl` 'method' function to it, thus allowing our kernel module / device driver to hook into this system call!

```
static struct file_operations iocnt_intf_fops = {
    .llseek = no_llseek,
    .ioctl = iocnt_intf_ioctl,
    [...]
};
```

Realistically, we shall have to figure out whether to use the `ioctl` or the `unlocked_ioctl` member of the `file_operations` structure depending on whether the module is running on Linux kernel version 2.6.36 or later or not; more on this follows.



In fact, adding new system calls to the kernel is not something one attempts lightly! The kernel chaps are *not* open to arbitrarily adding syscalls - it's a security-sensitive interface after all. More on this is documented here: <https://www.kernel.org/doc/html/latest/kernel-hacking/hacking.html#ioctls-not-writing-a-new-system-call>.

More on using the *ioctl* for interfacing follows, do read on.

Using the *ioctl* in user and kernel space

The `ioctl(2)` system call's signature is:

```
#include <sys/ioctl.h>
int ioctl(int fd, unsigned long request, ...);
```

The parameter list is a *varargs* - *variable arguments* - one. Realistically, we pass either two or three parameters typical to it:

- The first parameter is obvious - the file descriptor of the (in our case) device file that was opened.
- The second parameter, called `request`, is the interesting one: it's the *command* to be passed to the driver. In reality, it's an *encoding*, encapsulating a so-called *ioctl* 'magic number', a number and a 'type' (read/write), more on this soon follows.
- The (optional) third parameter, often called `argis` also an `unsigned long` quantity; we use it to either pass some data in the usual fashion to the underlying driver, or often, to return data to userspace by passing it's *address* and having the kernel write into it, utilizing C's so-called *value-result* or *in-out* parameter style.

Now, using the *ioctl* correctly is not as trivial as it is with many other APIs. Think about this for a moment: one can easily have a scenario where several userspace apps are issuing `ioctl(2)` system calls (with various 'commands' being issued) to their underlying device drivers. A problem becomes apparent: how will the kernel VFS layer direct the *ioctl* request to the 'correct' driver? Actually, think on this, the *ioctl* is typically performed on a char 'device file' that has a unique (*major*, *minor*) number; hence, how can another driver receive 'your' *ioctl* command (unless one intentionally, perhaps maliciously, sets up the device file(s) in just such a manner).

To achieve this safely and correctly, every application and driver defines a 'magic number' that will be encoded into all it's *ioctl* requests. The driver will first verify that every *ioctl* request it receives contains *it's* magic number; only then will it proceed to process it. This, of course, brings up an *ABI* - we need to allocate unique magic numbers (it could be a range) to each 'registered' driver; precisely this has been done. As this creates an *ABI*, the kernel folk document the same; you will find details on who is using which 'magic number' (or 'code') within the *ioctl* here: <https://www.kernel.org/doc/Documentation/ioctl/ioctl-number.txt>.

Wait, there's more: an *ioctl* request to the underlying driver can be one of essentially four things: a command to 'write' to the device, a command to 'read' from (or query) the device,

to do both read/write transfers, or neither. This information is (again) *encoded* into a request by defining certain bits to have meaning: to make this job easier, we have four helper macros that enable one to 'construct' `ioctl` 'commands':

- `_IO(type, nr)` : encode an `ioctl` command with no argument
- `_IOR(type, nr, datatype)` : encode an `ioctl` command for reading data from the kernel/driver
- `_IOW(type, nr, datatype)` : encode an `ioctl` command for writing data to the kernel/driver
- `_IOWR(type, nr, datatype)` : encode an `ioctl` command for read/write transfers

These macros are defined within the userspace `<sys/ioctl.h>` header and in the kernel here: `include/uapi/asm-generic/ioctl.h`. The accepted best practice is to create a *common header* file that defines the `ioctl` commands for an app/driver and include that file in both the usermode app as well as the device driver.

Here, as a demo, we shall design and implement a userspace app and a kernel-space device driver to drive a fictional 'device' that communicate via the `ioctl(2)` system call. Thus, we shall require to define some 'commands' to issue via the `ioctl` interface; we do so in a common header file, as seen below:

ch13/ioctl_intf/ioctl_llkd.h

```
/* The 'magic' number for our driver; see Documentation/ioctl/ioctl-
number.rst
 * Of course, we don't know for _sure_ if the magic # we choose here this
 * will remain free; it really doesn't matter, this is just for demo
 purposes; * don't try and upstream this without further investigation :-)
 */
#define IOCTL_LLKD_MAGIC          0xA8

#define IOCTL_LLKD_MAXIOCTL      3
/* our dummy ioctl (IOC) RESET command */
#define IOCTL_LLKD_IOCRESET      _IO(IOCTL_LLKD_MAGIC, 0)
/* our dummy ioctl (IOC) Query POWER command */
#define IOCTL_LLKD_IOCQPOWER    _IOR(IOCTL_LLKD_MAGIC, 1, int)
/* our dummy ioctl (IOC) Set POWER command */
#define IOCTL_LLKD_IOCSPower    _IOW(IOCTL_LLKD_MAGIC, 2, int)
```

We try and make the names we use in our macros meaningful. Our three 'commands' (highlighted in bold font above), are all prefixed with `IOCTL_LLKD_` indicating that all are `ioctl` commands for the *LLKD* project; next, they are suffixed with the format `IOC{Q|S}` for the *Query* or *Set* operation.

Now, let's see how we set things up at the code level from both the userspace as well as the kernel-space viewpoint.

Userspace - using the ioctl system call

The *userspace* signature of the `ioctl(2)` system call is:

```
#include <sys/ioctl.h>
int ioctl(int fd, unsigned long request, ...);
```

We can see that it takes a variable argument list; the arguments to the `ioctl` are:

- first parameter: the file descriptor of the file or device (as it will be in our case) to perform the `ioctl` operation upon (we 'get' the `fd` by performing an *open* on the device file)
- second parameter: the 'request' or 'command' being issued to the underlying device driver (or filesystem or whatever the `fd` represents)
- an optional third (or more) parameter(s): often, the third parameter is an integer (or a pointer to an integer or data structure); we use this method to either pass some additional information to the driver typically, when issuing a *set* kind of command, or to retrieve some information from the driver via the well understood *pass-by-reference* C paradigm, where we pass the pointer and have the driver 'poke' it, thus treating the parameter as, in effect, a return value



In effect, the `ioctl` is very often used as a *generic* system call. The use of the `ioctl` to perform 'command operations' on both hardware and software is almost embarrassingly large! We refer you to the kernel documentation (*Documentation/ioctl/<...>*) to see many actual examples. For example, you will find details on who is using which 'magic number' (or 'code') within the `ioctl` here: <https://www.kernel.org/doc/Documentation/ioctl/ioctl-number.txt>.

(Similarly, the `ioctl_list(2)` man page reveals the complete list of `ioctl` calls in the x86 kernel).

Below, we show snippets of the userspace 'C' application, particularly the issuing of the `ioctl(2)` system calls (for brevity and readability we leave out the error checking code; the full code is of course available on the book's GitHub repository):

ch13/ioctl_intf/userspace_ioctl/llkd_userspace.c

```
#include "../ioctl_llkd.h"
[...]
ioctl(fd, IOCTL_LLKD_IOCRESET, 0); // 1. reset the device
```

```

ioctl(fd, IOCTL_LLKD_IOCQPOWER, &power); // 2. query the 'power status'

// 3. Toggle it's power status
if (0 == power) {
    printf("%s: Device OFF, powering it On now ...\n", argv[0]);
    if (ioctl(fd, IOCTL_LLKD_IOCSPower, 1) == -1) { [...]
        printf("%s: power is ON now.\n", argv[0]);
    } else if (1 == power) {
        printf("%s: Device ON, powering it OFF in 3s ...\n", argv[0]);
        sleep(3); /* yes, careful here of sleep & signals! */
        if (ioctl(fd, IOCTL_LLKD_IOCSPower, 0) == -1) { [...]
            printf("%s: power OFF ok, exiting..\n", argv[0]);
        }
    }
}
[...]
```

How does our driver handle the userspace issued `ioctl`'s? We cover these details below.

Kernel space - using the `ioctl` system call

Above, we saw that the kernel driver will of course have to initialize it's `file_operations` structure to include the `ioctl` method. There is more to it though: the Linux kernel keep evolving of course; in early kernel versions, the developers had a very coarse granularity lock that, of course, hurt performance (don't worry, we discuss locking in detail in Ch 16 and Ch 17). It was so bad, it was nicknamed the *BKL - Big Kernel Lock!* The good news is that by kernel release 2.6.36, the developers got rid of this infamous lock. Doing so had some side effects though: one of them is that the number of parameters to the `ioctl` method within the kernel and thus within our `file_operations` data structure changed from 4 to 3 with the newer method - christened the `unlocked_ioctl`. Thus, for our demo driver, we do so with code like this when initializing our driver's `file_operations` structure:

ch13/ioctl_intf/kerneldrv_ioctl/llkd_kdrv.c:

```

#include "../ioctl_llkd.h"
#include <linux/version.h>
[...]
static struct file_operations ioctl_intf_fops = {
    .llseek = no_llseek,
    #if LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,36)
        .unlocked_ioctl = ioctl_intf_ioctl, // use the 'unlocked' version
    #else
        .ioctl = ioctl_intf_ioctl, // 'old' way
    #endif
};
```

Clearly, as it's defined within the driver 'fops', the `ioctl` is considered a private driver

interface (aka *driver-private*). Also, this same fact regarding the newer 'unlocked' version, has to be taken into account in it's definition within the driver code; our driver does so:

```
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,36)
static long ioctl_intf_ioctl(struct file *filp, unsigned int cmd, unsigned
long arg)
#else
static int ioctl_intf_ioctl(struct inode *ino, struct file *filp, unsigned
int cmd, unsigned long arg)
#endif
{
[...]
```

The key code here is the driver's *ioctl* method of course; think about it, once basic validity checks are done, all the driver really does is perform a *switch-case* on all possible valid *ioctl* 'commands' issued by the userspace app. Lets view the code below (for readability, we skip the `#if LINUX_VERSION_CODE >= ...` macro directive and just show the modern *ioctl* function signature, as well as some validity checks; please do view the full code from the book's GitHub repo for all details):

```
static long ioctl_intf_ioctl(struct file *filp, unsigned int cmd, unsigned
long arg)
{
    int retval = 0;
    MSG("In ioctl method, cmd=%d\n", _IOC_NR(cmd));

    /* Verify stuff: is the ioctl's for us? etc.. */
    [...]

    switch (cmd) {
    case IOCTL_LLKD_IOCRESET:
        MSG("In ioctl cmd option: IOCTL_LLKD_IOCRESET\n");
        /* ... Insert the code here to write to a control register to reset
           the device ... */
        break;
    case IOCTL_LLKD_IOCQPOWER: /* Get: arg is pointer to result */
        MSG("In ioctl cmd option: IOCTL_LLKD_IOCQPOWER\n"
            "arg=0x%x (drv) power=%d\n", (unsigned int)arg, power);
        if (!capable(CAP_SYS_ADMIN))
            return -EPERM;
        /* ... Insert the code here to read a status register to query the
           * power state of the device ... * here, imagine we've done that
           * and placed it into a variable 'power'
           */
        retval = __put_user(power, (int __user *)arg);
        break;
    case IOCTL_LLKD_IOCSPower: /* Set: arg is the value to set */
```

```

        if (!capable(CAP_SYS_ADMIN))
            return -EPERM;
        power = arg;
        /* ... Insert the code here to write a control register to set the
         * power state of the device ... */
        MSG("In ioctl cmd option: IOCTL_LLKD_IOCSPower\n"
            "power=%d now.\n", power);
        break;
    default:
        return -ENOTTY;
    }
    [...]

```

The `_IOC_NR` macro is used to extract the command number from the parameter `cmd`. Above, we can see that the driver 'reacts' to three valid cases of the `ioctl`:

- on receiving the `IOCTL_LLKD_IOCRESET` command, it performs a device 'reset'
- on receiving the `IOCTL_LLKD_IOCQPOWER` command, it 'queries' (Q for query) and returns the current 'power status' (by poking it's value into the third parameter `arg`, using the *value-result* 'C' programming approach)
- on receiving the `IOCTL_LLKD_IOCSPOWER` command, it 'sets' (S for set) the power status (to the value passed in the third parameter `arg`).

Of course, understand that here, as we're working with a purely fictional device, our driver does not actually perform any register (or other hardware) work. This driver is simply a template that one can make use of.

What if a hacker attempts to issue a command unknown to our driver in a clumsy 'hack'? Well, the initial validity checks will itself catch it; even if not, we shall hit the `default` case in our `ioctl` method, resulting in the driver returning `-ENOTTY` to userspace which will, via glibc 'glue' code, set `errno` to `ENOTTY`, informing it that the `ioctl` cannot be serviced. Our `perror(3)` API will display the *'Inappropriate ioctl for device'* error message. In fact, this is precisely what occurs what if a driver has *no* `ioctl` method (it's set to `NULL` in the `file_operations` structure) and a userspace app issues an `ioctl` against it.

I leave it to you to try out the above userspace/driver project example; for convenience, once the driver is loaded (via *insmod*), you can use the `ch13/userspace_ioctl/cr8devnode.sh` convenience script to generate the device file. Once setup, run the userspace app; you will find that running it in succession has the 'power state' of our fictional device get repeatedly toggled.



By the way, did you notice the usage of a `MSG()` macro in our driver code? It's merely a convenience, resulting in a `printk` only if the `DEBUG` macro is defined; doing it this way keeps the code clean by isolating the `#ifdef DEBUG ... #endif` kind of code from every instance of it being issued. The real work is done in our header file `convenient.h`. Similarly, there are several useful 'convenience' routines in this header; do take a look!

The ioctl as a debug interface

As mentioned at the beginning of this chapter, what about using the *ioctl* interface for debug purposes? Indeed, it can be used. You can always insert a 'debug' command into the *switch-case*; it can provide useful information to the userspace application on the driver status, the values of key variables (health monitoring too), etc. Not only that, think about it: unless explicitly documented to the end-user or customer, the precise commands used via the *ioctl* interface is unknown; thus, you are expected to document the interface providing sufficient detail for other teams or the customer, to make good use of them. This leads to an interesting point: you might choose to deliberately leave a certain *ioctl* command undocumented, it's now a 'hidden' command, that can be used by, say, field engineers to examine the device. (I leave doing this as an assignment to you, dear reader).



The kernel documentation on the *ioctl* includes this file: <https://www.kernel.org/doc/Documentation/ioctl/botching-up-ioctls.txt>. Though biased towards kernel graphics stack devs, it describes in quite some detail typical design mistakes, trade-offs and the like.

Fantastic, you're almost done! You have by now learned how to interface a kernel module (or other kernel component) with a usermode process or thread (within a userspace application) via various technologies. We began with *procfs*, then moved onto using *sysfs*, and then *debugfs*. The *netlink socket* and the *ioctl* system call completed the interfacing methods.

But with all this choice, which one do you actually use on a project? The next section helps you with this decision by providing a quick comparison between these various interfacing methods.

Comparing the interfacing methods - a table

Below, we attempt a quick comparison table of the various user-kernel interfacing methods

described in this chapter, based on a few parameters:

<i>Parameter / Interfacing method</i>	procfs	sysfs	debugfs	netlink socket	ioctl
<i>Ease of development</i>	easy to learn and use	(relatively) easy to learn and use	(very) easy to learn and use	harder; have to write userspace C plus driver code + understand socket APIs	fair/harder; have to write userspace C plus driver code
<i>Appropriate for what use</i>	core kernel; (a few older drivers may still use it)	device driver interfacing	driver (and other) interfacing for production and debug purposes	various interfacing: users include device drivers, core networking code, the udev system, etc	device driver interfacing mostly (includes many)
<i>Interface visibility</i>	visible to all; use permissions to control access	visible to all; use permissions to control access	visible to all; use permissions to control access	hidden from fs; doesn't pollute kernel namespace	hidden from fs; doesn't pollute kernel namespace
<i>Upstream kernel ABI for driver/module authors*</i>	usage in drivers is deprecated for mainline	the 'right way', the formally accepted approach to interface drivers with userspace	well supported and heavily used in mainline by drivers and other products	well supported (since 2.2)	well supported
<i>Useful for (driver) debugging purposes</i>	yes (although not supposed to in mainline)	no / not ideal	yes, very useful! 'no rules' design	no / not ideal	yes; (even) via 'hidden' commands

Table 13. : A brief comparison of various user <-> kernel interfacing technologies

* As mentioned earlier, the kernel community documents that *procfs*, *sysfs* and *debugfs* are all *ABIs*; their stability and lifespan isn't guaranteed. While that is the formal stance adopted by the community, the reality is that plenty of actual interfaces using these filesystems have become de-facto ones used by products in the real world. Nevertheless, we should follow the kernel community's 'rules' and guidelines regarding their usage.

Summary

In this chapter we covered an important aspect for kernel module / device driver authors - how exactly can one *interface between user and kernel space*. We walked you through several

interfacing methods; we began with an older one, interfacing via the venerable `proc` filesystem (and then mentioned why it's not a preferred method for driver authors). We then moved onto interfacing via the newer 2.6 based `sysfs`. This turns out to be *the* preferred interface to userspace, at least for a device driver. `Sysfs` has limitations though (recall the one-value-per-file `sysfs` 'rule'). Thus, using the completely free-format `debugfs` interfacing technique makes writing debug (and other) interfaces very simple and powerful indeed. The netlink socket is a powerful interfacing technology and is used by the network subsystem, `udev` and a few drivers; it does require some knowledge on socket programming and the kernel socket buffer. To perform generic 'command' operations on device drivers, the `ioctl` system call turns out to be a tremendous multiplexer and is often used by device driver authors (and other components) to interface with userspace.

Armed with this knowledge, you are now in a position to practically integrate your driver-level code with userspace applications (or scripts); often, a usermode GUI (Graphical User Interface) wants to display some values received from the kernel or device driver. You now know how the passing of these values from the kernel can be achieved.

So, good progress on our journey! In the next chapter we move onto some practical aspects of Linux kernel development - there, we shall learn how to setup delays, kernel timers, create and work with kernel threads, and use the kernel workqueue technology.

Questions

1. `sysfs_on_misc`: *sysfs assignment #1*:
Extend one of the 'misc' device drivers we wrote in *Ch 12*; setup two `sysfs` files and their read/write callbacks; test them from userspace
2. `sysfs_addrxlate`: *sysfs assignment #2 (a bit more advanced)*:
Address translation: exploiting the knowledge gained from this chapter and *Ch 7* section 'Direct-mapped RAM and address translation', write a simple platform driver that provides two `sysfs` interface files called `addrxlate_kva2pa` and `addrxlate_pa2kva`; the way it should work: writing a *kva* (kernel virtual address) into the `sysfs` file `addrxlate_kva2pa` should have the driver read and translate the *kva* into it's corresponding *physical address* (*pa*); then reading from the same file should cause the *pa* to be displayed. Vice-versa with the `addrxlate_pa2kva` `sysfs` file
3. `dbgfs_disp_pgoff`: *debugfs assignment #1*:
Write a kernel module that sets up a `debugfs` file here: `<debugfs_mount_point>/dbgfs_disp_pgoff` ; when read, it should display (to userspace) the current value of the kernel macro `PAGE_OFFSET`

4. `dbgfs_showall_threads` : *debugfs assignment #2* :

Write a kernel module that sets up a `debugfs` file here:

`<debugfs_mount_point>/dbgfs_showall_threads/dbgfs_showall_threads`; when read, it should display some attributes of every thread alive. (Similar to our code here: `ch6/foreach/thrd_showall`; there, though, the threads are displayed *only* at `insmod` time; with a `debugfs` file, you can display info on all threads at any time you choose to)!

Suggested output is CSV format:

`TGID, PID, current, stack-start, name, #threads`

([name] in square brackets => kernel thread;

#threads only displays a positive integer; no output implies a single-threaded process;

f.e.:

`130, 130, 0xfffff9f8b3cd38000, 0xfffffc13280420000, [watchdogd]`

5. *ioctl assignment #1*: Using the provided `ch13/ioctl_intfl` code as a template, write a userspace 'C' application and a kernel-space (char) device driver implementing the *ioctl* method. Add an *ioctl* command `IOCTL_LLKD_IOCQPGOFF` to return the value of `PAGE_OFFSET` (within the kernel) to userspace6. *ioctl undoc*: *ioctl assignment #2*:

Using the provided `ch13/ioctl_intfl` code as a template, write a userspace 'C' application and a kernel-space (char) device driver implementing the *ioctl* method. Add in a 'driver context' data structure (that we use in several examples), allocate and initialize it. Now, in addition to the earlier three *ioctl* 'commands' we use, setup a fourth 'undocumented' command (you can call it `IOCTL_LLKD_IOCQDRVSTAT`). It's job: when queried from userspace via `ioctl(2)`, it must return the contents of the 'driver context' data structure to userspace; the userspace 'C' app must print out the current content of every member of that structure.

Further Reading

- FYI, some information on using the very common I2C protocol within a Linux device driver
 - Article on I2C protocol basics: 'How to use I2C in STM32F103C8T6? STM32 I2C Tutorial', Mar 2020
 - Kernel documentation: Implementing I2C device drivers : <https://www.kernel.org/doc/html/latest/i2c/writing-clients.html>

To aid you delve deeper into the subject with useful materials, we provide a rather detailed

list of online references, links (and at times even books) in a *Further Reading* markdown document - organized chapter-wise - on this book's *GitHub* repository. The *Further Reading* document is available here: https://github.com/PacktPublishing/Learn-Linux-Kernel-Development/blob/master/Further_Reading.md.

Index