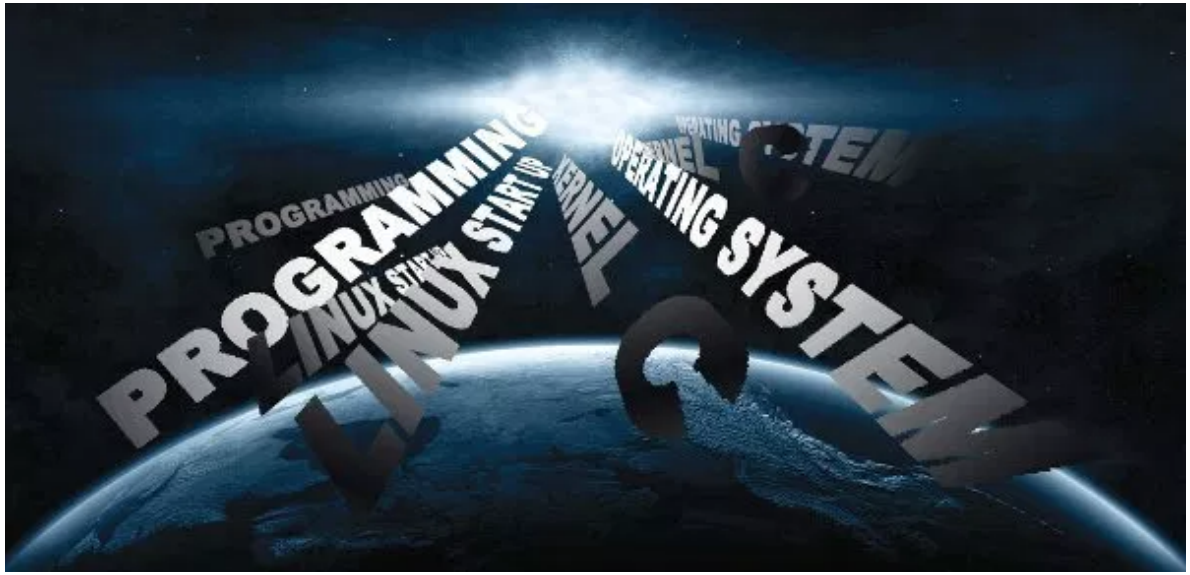
Device Drivers, Part 3: Kernel C Extras in a Linux Driver

By **Anil Kumar Pugalia** - January 1, 2011



This article in the series on Linux device drivers deals with the kernel's message logging, and kernel-specific GCC extensions Enthused by how Pugs impressed their professor in the last class, Shweta wanted to do so too. And there was soon an opportunity: finding out where the output of printk had gone. So, as soon as she entered the lab, she grabbed the best system, logged in, and began work. Knowing her professor well, she realised that he would have dropped a hint about the possible solution in the previous class itself. Going over what had been taught, she remembered the error output demonstration from `insmod vfat.ko` — running `dmesg | tail` . She immediately tried that, and found the `printk` output there.

But how did it come to be here? A tap on her shoulder roused her from her thoughts. "Shall we go for a coffee?" proposed Pugs.

"But I need to –".

"I know what you're thinking about," interrupted Pugs. "Let's go, I'll explain you all about dmesg."

Kernel message logging

Over coffee, Pugs began his explanation.

As far as parameters are concerned, `printf` and `printk` are the same, except that when programming for the kernel, we don't bother about the float formats `%f`, `%lf` and the like. However, unlike `printf`, `printk` is not designed to dump its output to some console.

In fact, it cannot do so; it is something in the background, and executes like a library, only when triggered either from hardware-space or user-space. All `printk` calls put their output into the (log) ring buffer of the kernel. Then, the `syslog` daemon running in user-space picks them up for final processing and redirection to various devices, as configured in the configuration file `/etc/syslog.conf`.

You must have observed the out-of-place macro `KERN_INFO`, in the `printk` calls, in the last article. That is actually a constant string, which gets concatenated with the format string after it, into a single string. Note that there is no comma (,) between them; they are not two separate arguments. There are eight such macros defined in `linux/kernel.h` in the kerne source, namely:

```
#define KERN_EMERG "<0>"   /* system is unusable               */
#define KERN_ALERT "<1>"   /* action must be taken immediately    */
#define KERN_CRIT "<2>"    /* critical conditions     */
#define KERN_ERR "<3>"     /* error conditions            */
#define KERN_WARNING "<4>" /* warning conditions      */
#define KERN_NOTICE "<5>"  /* normal but significant condition    */
#define KERN_INFO "<6>"    /* informational           */
#define KERN_DEBUG "<7>"   /* debug-level messages        */
```

Now depending on these log levels (i.e., the first three characters in the format string), the `syslog` user-space daemon redirects the corresponding messages to their configured locations. A typical destination is the log file `/var/log/messages`, for

all log levels. Hence, all the `printk` outputs are, by default, in that file. However, they can be configured differently — to a serial port (like `/dev/ttyS0`), for instance, or to all consoles, like what typically happens for `KERN_EMERG`.

Now, `/var/log/messages` is buffered, and contains messages not only from the kernel, but also from various daemons running in user-space. Moreover, this file is often not readable by a normal user. Hence, a user-space utility, `dmesg`, is provided to directly parse the kernel ring buffer, and dump it to standard output. Figure 1 shows snippets from the two.

Figure 1: Kernel's message logging

Kernel-specific GCC extensions

Shweta, frustrated since she could no longer show off as having discovered all these on her own, retorted, "Since you have explained all about printing in the kernel, why don't you also tell me about the weird C in the driver as well — the special keywords `__init` , `__exit` , etc."

These are not special keywords. Kernel C is not "weird C", but just standard C with some additional extensions from the C compiler, GCC. Macros `__init` and `__exit` are just two of these extensions. However, these do not have any relevance in case we are using them for a dynamically loadable driver, but only when the same code gets built into the kernel. All functions marked with `__init` get placed inside the `init` section of the kernel image automatically, by GCC, during kernel compilation; and all functions marked with `__exit` are placed in the `exit` section of the kernel image.

What is the benefit of this? All functions with `__init` are supposed to be executed only once during bootup (and not execu again till the next bootup). So, once they are executed during bootup, the kernel frees up RAM by removing them (by freeir the `init` section). Similarly, all functions in the `exit` section are supposed to be called during system shutdown.

Now, if the system is shutting down anyway, why do you need to do any cleaning up? Hence, the `exit` section is not even loaded into the kernel — another cool optimisation. This is a beautiful example of how the kernel and GCC work hand-in-hai to achieve a lot of optimisation, and many other tricks that we will see as we go along. And that is why the Linux kernel can only be compiled using GCC-based compilers — a closely knit bond.

The kernel function's return guidelines

While returning from coffee, Pugs kept praising OSS and the community that's grown around it. Do you know why different individuals are able to come together and contribute excellently without any conflicts, and in a project as huge as Linux, at that? There are many reasons, but most important amongst them is that they all follow and abide by inherent coding guidelines.

Take, for example, the kernel programming guideline for returning values from a function. Any kernel function needing error handling, typically returns an integer-like type — and the return value again follows a guideline. For an error, we return a negative number: a minus sign appended with a macro that is available through the kernel header `linux/errno.h`, that includes the various error number headers under the kernel sources — namely, `asm/errno.h`, `asm-generic/errno.h`, `asm-generic/errno-base.h`.

For success, zero is the most common return value, unless there is some additional information to be provided. In that case, a positive value is returned, the value indicating the information, such as the number of bytes transferred by the function.

Kernel C = pure C

Once back in the lab, Shweta remembered their professor mentioning that no `/usr/include` headers can be used for kerne programming. But Pugs had said that kernel C is just standard C with some GCC extensions. Why this conflict?

Actually this is not a conflict. Standard C is pure C — just the language. The headers are not part of it. Those are part of the standard libraries built in for C programmers, based on the concept of reusing code.

Does that mean that all standard libraries, and hence, all ANSI standard functions, are not part of "pure" C? Yes, that's righ Then, was it really tough coding the kernel?

Well, not for this reason. In reality, kernel developers have evolved their own set of required functions, which are all part of the kernel code. The `printk` function is just one of them. Similarly, many string functions, memory functions, and more, are all part of the kernel source, under various directories like `kernel`, `ipc`, `lib`, and so on, along with the corresponding headers under the `include/linux` directory.

"Oh yes! That is why we need to have the kernel source to build a driver," agreed Shweta.