The Linux Kernel Driver Interface

(all of your questions answered and then some)

Greg Kroah-Hartman <greg@kroah.com>

This is being written to try to explain why Linux **does not have a binary kernel interface, nor does it have a stable kernel interface**.

> ### *Note*
>
> Please realize that this article describes the **in kernel** interfaces, not the kernel to userspace interfaces.
>
> The kernel to userspace interface is the one that application programs use, the syscall interface. That interface is **very** stable over time, and will not break. I have old programs that were built on a pre 0.9something kernel that still work just fine on the latest 2.6 kernel release. That interface is the one that users and application programmers can count on being stable.

Executive Summary

You think you want a stable kernel interface, but you really do not, and you don't even know it. What you want is a stable running driver, and you get that only if your driver is in the main kernel tree. You also get lots of other good benefits if your driver is in the main kernel tree, all of which has made Linux into such a strong, stable, and mature operating system which is the reason you are using it in the first place.

Intro

It's only the odd person who wants to write a kernel driver that needs to worry about the in-kernel interfaces changing. For the majority of the world, they neither see this interface, nor do they care about it at all.

First off, I'm not going to address **any** legal issues about closed source, hidden source, binary blobs, source wrappers, or any other term that describes kernel drivers that do not have their source code released under the GPL. Please consult a lawyer if you have any legal questions, I'm a programmer and hence, I'm just going to be describing the technical issues here (not to make light of the legal issues, they are real, and you do need to be aware of them at all times.)

So, there are two main topics here, binary kernel interfaces and stable kernel source interfaces. They both depend on each other, but we will discuss the binary stuff first to get it out of the way.

Binary Kernel Interface

Assuming that we had a stable kernel source interface for the kernel, a binary interface would naturally happen too, right? Wrong. Please consider the following facts about the Linux kernel:

- Depending on the version of the C compiler you use, different kernel data structures will contain different alignment of structures, and possibly include different functions in different ways (putting functions inline or not.) The individual function organization isn't that important, but the different data structure padding is very important.

- Depending on what kernel build options you select, a wide range of different things can be assumed by the kernel:

    - different structures can contain different fields

- Some functions may not be implemented at all, (i.e. some locks compile away to nothing for non-SMP builds.)

- Memory within the kernel can be aligned in different ways, depending on the build options.

- Linux runs on a wide range of different processor architectures. There is no way that binary drivers from one architecture will run on another architecture properly.

Now a number of these issues can be addressed by simply compiling your module for the exact specific kernel configuration, using the same exact C compiler that the kernel was built with. This is sufficient if you want to provide a module for a specific release version of a specific Linux distribution. But multiply that single build by the number of different Linux distributions and the number of different supported releases of the Linux distribution and you quickly have a nightmare of different build options on different releases. Also realize that each Linux distribution release contains a number of different kernels, all tuned to different hardware types (different processor types and different options), so for even a single release you will need to create multiple versions of your module.

Trust me, you will go insane over time if you try to support this kind of release, I learned this the hard way a long time ago...

Stable Kernel Source Interfaces

This is a much more "volatile" topic if you talk to people who try to keep a Linux kernel driver that is not in the main kernel tree up to date over time.

Linux kernel development is continuous and at a rapid pace, never stopping to slow down. As such, the kernel developers find bugs in current interfaces, or figure out a better way to do things. If they do that, they then fix the current interfaces to work better. When they do so, function names may change, structures may grow or shrink, and function parameters may be reworked. If this happens, all of the instances of where this interface is used within the kernel are fixed up at the same time, ensuring that everything continues to work properly.

As a specific examples of this, the in-kernel USB interfaces have undergone at least three different reworks over the lifetime of this subsystem. These reworks were done to address a number of different issues:

- A change from a synchronous model of data streams to an asynchronous one. This reduced the complexity of a number of drivers and increased the throughput of all USB drivers such that we are now running almost all USB devices at their maximum speed possible.

- A change was made in the way data packets were allocated from the USB core by USB drivers so that all drivers now needed to provide more information to the USB core to fix a number of documented deadlocks.

This is in stark contrast to a number of closed source operating systems which have had to maintain their older USB interfaces over time. This provides the ability for new developers to accidentally use the old interfaces and do things in improper ways, causing the stability of the operating system to suffer.

In both of these instances, all developers agreed that these were important changes that needed to be made, and they were made, with relatively little pain. If Linux had to ensure that it will preserve a stable source interface, a new interface would have been created, and the older, broken one would have had to be maintained over time, leading to extra work for the USB developers. Since all Linux USB developers do their work on their own time, asking programmers to do extra work for no gain, for free, is not a possibility.

Security issues are also very important for Linux. When a security issue is found, it is fixed in a very short amount of time. A number of times this has caused internal kernel interfaces to be reworked to prevent the security problem from occurring. When this happens, all drivers that use the interfaces were also fixed at the same time, ensuring that the security problem was fixed and could not come back at some future time accidentally. If the internal interfaces were not allowed to change, fixing this kind of security problem and insuring that it could not happen again would not be possible.

Kernel interfaces are cleaned up over time. If there is no one using a current interface, it is deleted. This ensures that the kernel remains as small as possible, and that all potential interfaces are tested as well as they can be (unused interfaces are pretty much impossible to test for validity.)

What to do

So, if you have a Linux kernel driver that is not in the main kernel tree, what are you, a developer, supposed to do? Releasing a binary driver for every different kernel version for every distribution is a nightmare, and trying to keep up with an ever changing kernel interface is also a rough job.

Simple, get your kernel driver into the main kernel tree (remember we are talking about GPL released drivers here, if your code doesn't fall under this category, good luck, you are on your own here, you leech <insert link to leech comment from Andrew and Linus here>.) If your driver is in the tree, and a kernel interface changes, it will be fixed up by the person who did the kernel change in the first place. This ensures that your driver is always buildable, and works over time, with very little effort on your part.

The very good side effects of having your driver in the main kernel tree are:

- The quality of the driver will rise as the maintenance costs (to the original developer) will decrease.
- Other developers will add features to your driver.
- Other people will find and fix bugs in your driver.
- Other people will find tuning opportunities in your driver.
- Other people will update the driver for you when external interface changes require it.
- The driver automatically gets shipped in all Linux distributions without having to ask the distros to add it.

As Linux supports a larger number of different devices "out of the box" than any other operating system, and it supports these devices on more different processor architectures than any other operating system, this proven type of development model must be doing something right :)