

# Linux Kernel Communication Methods:

## Summary [What is this Document]:

In my limited exploits with Linux kernel development, I came across one program design issue that plagued me for quite a while and I tried several different solutions for it. That design issue was how to get data from the kernel to user space on linux and in some cases also from user space back to the kernel, or in other words single-direction communication or bi-directional communication to/from the linux kernel. For those who are just starting out with the kernel here's a description of the problem (feel free to skip it if you wish):


The Linux kernel is considered a closed and protected space to ensure that the kernel doesn't crash even in the most extreme cases. In other words, the kernel should always be separated from any outside influence. Those around us always influence our behavior, after all. To that end, the kernel handles hardware devices, time schedule, and a list of system services, but anything outside of its direct implementation is "user space." Code written and compiled to run in user space doesn't use kernel data structures, kernel devices, or kernel memory. To get info from the kernel, often user space programs must open character devices stored in /dev and /sys and read those in read-only mode, for instance. When two-way communication is required, it's generally for a kernel module loaded as an extension to general kernel behavior, and it is a rare design method and the lack of examples that show it can be daunting. The main kernel generally only accepts calls from user space via system calls of various types, and as a general rule, typically only modules use methods outside of system calls to communicate with user space.

A kernel module is a device driver, or driver for a system service, such as a new filesystem, that's loaded into the kernel as an expansion module. Modules are required to follow very closely defined setup criteria to compile using existing kernel code. A module can be loaded with the command insmod into the running linux kernel, and removed with rmmod. Inside the module's code, you generally will define the system calls necessary to begin and end the module and communicate with the kernel. These system calls are, for instance, sometimes interrupts generated by hardware devices the module acts as a driver for, or they can be system interfaces such as the linux vfs layer which acts as a go-between for filesystems to talk to the kernel. Using these methods, a module can act as part of the system without directly altering kernel code, and often a module may have layers of protection between itself and most of the running kernel code.

So, with everything geared toward protecting the kernel, it can sometimes be challenging to get data out and into the secure areas without delays, bottlenecks, bugs, and outright roadblocks. This document assumes that the reader may (or may not) be a beginner kernel developer and is looking for some method of communicating data to or from the Linux kernel. It goes over several methods of communication, expounds on their good and bad points, notes what situations are good or bad uses for that method, and where possible example code is given that can get a new developer up and running. One area this doc pays special attention to is the speed/performance of these communication methods and what data profiles they would best fit.

## The Legend of communication Traits:

In my experience, it's easy to find documentation on kernel communication, but nearly impossible to know when each method is/is not a good fit. To that end, this document is arranged with methods of communicating small messages at the beginning and progresses to the better methods of communicating large amounts of data later in the document. Further, each method is marked with several icons which will make it obvious what each communication method is good at with a glance. This section provides the legend for these icons to allow the reader to skim the methods quickly.

Single-direction Communication: 

Two-way Communication: 

Amount of Data the method is intended for:



Speed:



## Methods:

**Perror and other kernel macros.**  

Perror, printk, and other such tools are methods of logging data to syslog from the kernel. In other words, they're a way of passing a message from the kernel ring buffer used for logging into a user-space program, like syslog or rsyslog, that picks up the message and delivers it to the right log file. This method is limited to a single use. It can take kernel module info and deliver it to a file via user space in a line-by-line format. This can be useful at times if you only want to transfer single lines (or formatted lines) into a file that can be scraped and processed from a user-space program. It's the simplest method by far to export info from the kernel.

Good:

1. Extremely simplistic.

2. Perfect for formatted lines that can be easily picked up by a program scraping the log in user-space.

Bad:

1. Too limited for most realistic uses.
2. Limited to a line-by-line format.
3. The same data also appears in a logging file, so it shouldn't be used for private data.

A line is exported simply by using the kernel's `printk` command, which uses the same basic syntax as `printf` in user-space for the most part. Without any further config, that line will show up in `/var/log/messages` or `dmesg`. However, if you add a tag that describes your module like this:

```
printk(KERN_INFO "<your_module_name>: .....normal message line.....");
```

You can then edit the system's `syslog.conf` or `rsyslog.conf` file and add your tag to ensure lines which use it go to a separate file. For instance, this line would direct lines output by the named kernel module to a log file with the same name:

```
your_module_name.*                                /var/log/your_module_name.log
```

This is a highly limited method of transferring data from the kernel to user space but given how complicated it is to work with the kernel, the flexibility of this method isn't to be underestimated. Also, it should be noted that transfers to user-space would be fast, but since the file must be scraped by a user-space program, that would cause significant slow-down, depending on what the data is required for. For instance, if the intent is to wake up a user-space thread whenever certain "things" happen in a module, then this method would be -perfect. The data would arrive within a timely manner and by using a file poll in user-space, you could read new data instantly. However, if your goal is to transfer a large block of data or several data points periodically, then the slow-down in scraping large amounts of data would be punishing.

## **Netlink**

I basically made a label here so I could tell you that under no circumstances should you ever consider using this one. There are two versions of Netlink, and this is not the one to use. See below for the explanation and the right way to do this.

## **Netlink Generic**

When you search for netlink on the web, you should use the term "netlink generic". Netlink (not generic) is a general protocol for kernel communication that gives 30 standard IDs for system communication, and those 30 are allocated to kernel systems, so don't touch them. You can. But don't. You're going to eventually run into conflicts with other kernel modules, and that's a very bad idea. The original Netlink

was devices years ago and was limited to the most common modules used with linux kernels in the beginning. Luckily, several years after the original Netlink release the kernel developers realized this was a concept with limited scope and viability and released an update that included Netlink Generic.

Netlink Generic allows the protocol to be used with a service name rather than a simple integer from 1 to 30. So, unless you're a module developer that owns one of the 30 golden module ID's, Netlink Generic is the only netlink that exists. 30? Really? These day every device has an embedded version of linux it seems, and there are literally automated pet feeders for dogs and cats that have more than 30 modules running.

Good:

1. Netlink is fast.
2. It's standard and intended for messaging systems mainly. So, for logging especially it's probably the right choice, despite its annoying bugs.

Bad:

1. It's standard. It's inflexible and basically impossible to make extensible without some serious coding and trouble.

### **Shared Memory**

The gotcha here and in the sections below is that there are many ways to use shared memory to pass information. When looking for information on the web about kernel communication, one of the issues you'll run into is distinguishing which type a particular example is using, which an author might be talking about, and trying to pull apart the spaghetti of information available on shared memory. To further complicate that topic, you would generally need some type of lock on the memory to use it, and there are at least 3 main types that are typically used in Linux. To make this topic easier, we'll first talk about shared memory, and its uses in kernel communication. Then later, we'll introduce locks and how to protect your shared memory access. This is especially helpful since your method of sharing data may determine which lock types you can use. All of these methods have the same basic advantages and disadvantages.

Good:

1. Shared memory is fast, arguably probably the fastest communication method.
2. It's also very flexible in the message types it can transfer. Nearly anything imaginable can be transferred through memory.

Bad:

2. It's standard. It's inflexible and basically impossible to make extensible without some serious coding and trouble.

What we're generally talking about when sharing memory between the kernel and user space, however, is using a shared Memory block directly by mapping in such a way that both spaces can see and address it. That narrows the scope of the methods to create one somewhat.

It's worth pointing out before leaving this section that using a mutex to open a simple shared memory block with `shm_open` is not supported for communication between kernel and user space. This is generally the method for pthreads to share a block of memory. The kernel uses a separate addressing space from user memory, however. The methods above work to get around this addressing issue. Using shared memory directly without the methods above does not work around the issue.

## Character and Block Devices

Character and Block devices are communication pipes setup under the `/dev` directory under Linux. However, don't be fooled. These are not actually files and are not subject to the delay of using the VFS filesystem layer in Linux. Remember, in Unix anything can be a file. In this case, even a memory segment set aside for communication can be a file. These are the same thing as the shared memory spoken of above. However, there are many ways to access shared memory, and if it has a `/dev` file associated with it, then one of those ways is by simply opening it like a file from user space and reading/writing it, then closing it. This is meant to be a super-simple communication method that treats the memory like a file and eliminates the complexity for user-space developers. The catch here, from a module developer perspective, is that this method is a little mind-bending. All reads and writes of kernel data is initiated from the user side of the communication in almost all cases for any communication method. `/dev` devices is no exception. So, if a kernel module wishes to communicate something to a user space program, then it must initiate an interrupt or set a poll to notify user space to open the `/dev` file. This is not difficult at all to do; however, it has an unfortunate side-effect. From a kernel perspective, writing data to user space requires a user program to open the file and read data, so for the module developer, a write to user space is a read. Conversely, a read from module space is a write. For the person developing the module, these terms are exactly opposite of what you mentally think they should be. I'm not sure about other kernel developers, but I remember this being particularly brain-deforming for me when I began module developing.

There are 3 basic steps to creating a character or block device. And the main step of allocating the memory, strangely, doesn't necessarily need to be your first move.

```
struct file_operations cmd_fops = {
    .owner = THIS_MODULE,
    .read = hi_comm_cmd_device_read,
    .write = hi_comm_cmd_device_write,
    .release = hifs_comm_device_release,
    .poll = hifs_cmd_device_poll,
};
```

```
    major = register_chrdev(0, DEVICE_FILE_BLOCK "_block",
&block_fops);
    if (major < 0) {
        pr_err("hivefs: Failed to register block (file) comm queue
device\n");
        return major;
    }
```

```
#if LINUX_VERSION_CODE >= KERNEL_VERSION(6, 4, 0)
    dev_class = class_create(DEVICE_FILE_INODE);
    device_create(dev_class, NULL, MKDEV(major, 0), NULL,
DEVICE_FILE_INODE);
    dev_class = class_create(DEVICE_FILE_BLOCK);
    device_create(dev_class, NULL, MKDEV(major, 1), NULL,
DEVICE_FILE_BLOCK);
    dev_class = class_create(DEVICE_FILE_CMDS);
    device_create(dev_class, NULL, MKDEV(major, 2), NULL,
DEVICE_FILE_CMDS);
#else
    dev_class = class_create(THIS_MODULE, DEVICE_FILE_INODE);
    device_create(dev_class, NULL, MKDEV(major, 0), NULL,
DEVICE_FILE_INODE);
    dev_class = class_create(THIS_MODULE, DEVICE_FILE_BLOCK);
    device_create(dev_class, NULL, MKDEV(major, 1), NULL,
DEVICE_FILE_BLOCK);
    dev_class = class_create(THIS_MODULE, DEVICE_FILE_CMDS);
```

```
    device_create(dev_class, NULL, MKDEV(major, 2), NULL,  
DEVICE_FILE_CMDS);  
#endif
```

```
unregister_chrdev(major, DEVICE_FILE_INODE);  
unregister_chrdev(major, DEVICE_FILE_BLOCK "_block");  
unregister_chrdev(major, DEVICE_FILE_CMDS);
```

#### RelayFS (Now name changed to Relay Interface)

Relay is an interface that does one-way communication only, from the running kernel to user space. It is the module which sets up the /sys directory in Linux and under there are files maintained by various modules which can be opened only for reading. The files contain whatever information the module deems it wishes to communicate with user space. If you're looking at a one-way communication method and your data can be easily parse-able in a file, then this is probably the method you want.

#### Atomic variables

I love atomic variables, and I wish there were a way in the kernel to create atomic variables from a struct like it done in user space by mmaping a struct-sized block and then using shm\_open to access it. If there were such a method, this entire document would most likely be irrelevant. Unfortunately, an atomic variable can only be a very simple variable. It is a form of shared memory, however, that automatically manages its own locks and ensures that only one thread can read or write to it at a time. To share it with user space from the kernel, you would use a device or character file as described above. Then, just like any file, it can be opened and read/written from user space with the standard file commands we've already given examples for. However, it can only be used for known integer types, chars, or char arrays. Never anything more complicated than that, simply because if you try to read typedef variables like a file, there is generally reserved bits/bytes within the variable boundaries that are blank and variable length. In other words, in a struct like this:

```
struct my_struct {  
    int a;
```

```
    int b;  
} example_struct;
```

We have no way of telling from this struct where the integer a was put into memory when instantiated and what boundaries it may have around it. Because of that, we don't know if there is a zero-length separation between variable a and b, or if there is a 40 byte separation between the two. So, if we try to read that struct like a file, parsing it may be impossible. The more complicated the typedef, the worse that problem gets. It's even more impossible if the programmer doesn't zero out every variable they create, because then there may be space with garbage between the two integers. So, Atomics by necessity must be simple.

## File Read/Write

### Protecting Shared Memory and Files

There are many types of locks that can be used from both kernel and user space. Whether you're using shared memory to pass information or a file, the idea would be to use your lock in two ways. First, it protects one side of the communication while the other side is writing. Data corruption would result if both sides were out of sync with who was reading or writing to the file. The second way is that most lock types can be used to notify user space when the kernel is ready for either reading or writing data. Using these two ways to manage your file, you will always be aware of which side is waiting, and who expects a read or write. That way, your communication never gets out of sync, which would result in one or both sides sending or receiving the wrong data and data corruption or a program crash potentially being the worst-case scenario. Also, there's a separate system service in addition to locks that can help enhance them. By using a wait queue, the queue can tell you when someone is waiting on an operation and then send an interrupt when it's time for that thread to access the communication device. This service can be used with almost any type of lock, and since locks generally only tell you that memory or the file is in use, the wait queue enhances that information greatly by allowing you to pick which thread accesses the file and even using multiple threads in the queue to choose order.

### Spinlocks



Semaphores

Mutexes

Mutexes

Wait Queue