**Linux Character Drivers**

[online book](http://lwn.net/Kernel/LDD3/),

*Linux Device Drivers* by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman.

W’s of character drivers

We already know what drivers are, and why we need them. What is so special about character drivers? If we write drivers for byte-oriented operations (or, in C lingo, character-oriented operations), then we refer to them as character drivers. Since the majority of devices are byte-oriented, the majority of device drivers are character device drivers.

Take, for example, serial drivers, audio drivers, video drivers, camera drivers, and basic I/O drivers. In fact, all device drivers that are neither storage nor network device drivers are some type of a character driver. Let’s look into the commonalities of these character drivers, and how Shweta wrote one of them.

The complete connection

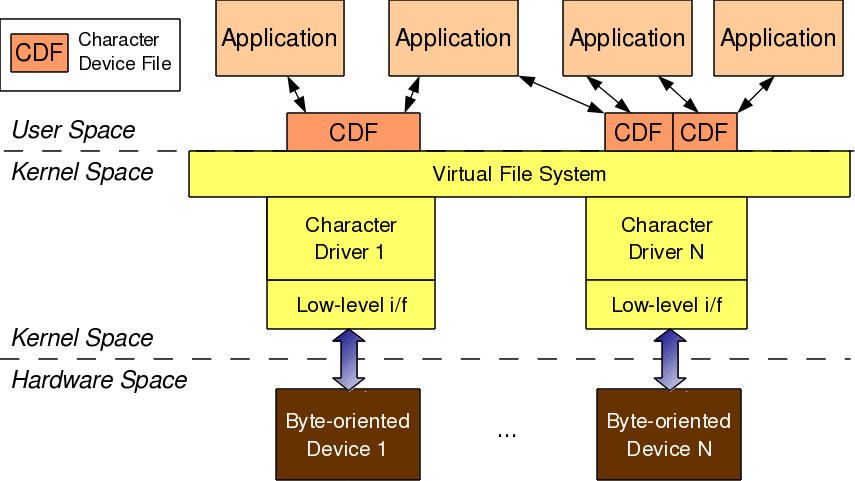


Figure 1: Character driver overview

As shown in Figure 1, for any user-space application to operate on a byte-oriented device (in hardware space), it should use the corresponding character device driver (in kernel space). Character driver usage is done through the corresponding character device file(s), linked to it through the virtual file system (VFS). What this means is that an application does the usual file operations on the character device file. Those operations are translated to the corresponding functions in the linked character device driver by the VFS. Those functions then do the final low-level access to the actual device to achieve the desired results.

Note that though the application does the usual file operations, their outcome may not be the usual ones. Rather, they would be as driven by the corresponding functions in the device driver. For example, a write followed by a read may not fetch what has just been written to the character device file, unlike for regular files. Remember that this is the usual expected behaviour for device files. Let’s take an audio device file as an example. What we write into it is the audio data we want to play back, say through a speaker. However, the read would get us audio data that we are recording, say through a microphone. The recorded data need not be the played-back data.

In this complete connection from the application to the device, there are four major entities involved:

1. Application
2. Character device file
3. Character device driver
4. Character device

The interesting thing is that all of these can exist independently on a system, without the other being present. The mere existence of these on a system doesn’t mean they are linked to form the complete connection. Rather, they need to be explicitly connected. An application gets connected to a device file by invoking the open system call on the device file.

Device file(s) are linked to the device driver by specific registrations done by the driver. The driver is linked to a device by its device-specific low-level operations. Thus we form the complete connection. With this, note that the character device file is not the actual device, but just a place-holder for the actual device.

Major and minor numbers

The connection between the application and the device file is based on the name of the device file. However, the connection between the device file and the device driver is based on the number of the device file, not the name. This allows a user-space application to have any name for the device file, and enables the kernel-space to have a trivial index-based linkage between the device file and the device driver. This device file number is more commonly referred to as the <major, minor> pair, or the major and minor numbers of the device file.

Earlier (till kernel 2.4), one major number was for one driver, and the minor number used to represent the sub-functionalities of the driver. With kernel 2.6, this distinction is no longer mandatory; there could be multiple drivers under the same major number, but obviously, with different minor number ranges.

However, this is more common with the non-reserved major numbers, and standard major numbers are typically preserved for single drivers. For example, 4 for serial interfaces, 13 for mice, 14 for audio devices, and so on. The following command would list the various character device files on your system:

|  |
| --- |
| $ ls -l /dev/ | grep "^c" |

<major, minor> related support in kernel 2.6

Type (defined in kernel header linux/types.h):

* dev\_t contains both major and minor numbers

Macros (defined in kernel header linux/kdev\_t.h):

* MAJOR(dev\_t dev) extracts the major number from dev
* MINOR(dev\_t dev) extracts the minor number from dev
* MKDEV(int major, int minor) creates the dev from major and minor.

Connecting the device file with the device driver involves two steps:

1. Registering for the <major, minor> range of device files.
2. Linking the device file operations to the device driver functions.

The first step is achieved using either of the following two APIs, defined in the kernel header linux/fs.h:

|  |
| --- |
| + int register\_chrdev\_region(dev\_t first, unsigned int cnt, char \*name);  + int alloc\_chrdev\_region(dev\_t \*first, unsigned int firstminor, unsigned int cnt, char \*name); |

The first API registers the cnt number of device file numbers, starting from first, with the given name. The second API dynamically figures out a free major number, and registers the cnt number of device file numbers starting from <the free major, firstminor>, with the given name. In either case, the /proc/devices kernel window lists the name with the registered major number. With this information, Shweta added the following into the first driver code:

|  |
| --- |
| #include <linux/types.h>  #include <linux/kdev\_t.h>  #include <linux/fs.h>    static dev\_t first; // Global variable for the first device number |

In the constructor, she added:

|  |
| --- |
| if (alloc\_chrdev\_region(&first, 0, 3, "Shweta") < 0)  {      return -1;  }  printk(KERN\_INFO "<Major, Minor>: <%d, %d>\n", MAJOR(first), MINOR(first)); |

In the destructor, she added:

|  |
| --- |
| unregister\_chrdev\_region(first, 3); |

It’s all put together, as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | #include <linux/module.h>  #include <linux/version.h>  #include <linux/kernel.h>  #include <linux/types.h>  #include <linux/kdev\_t.h>  #include <linux/fs.h>    static dev\_t first; // Global variable for the first device number    static int \_\_init ofcd\_init(void) /\* Constructor \*/  {      printk(KERN\_INFO "Namaskar: ofcd registered");      if (alloc\_chrdev\_region(&first, 0, 3, "Shweta") < 0)      {          return -1;      }      printk(KERN\_INFO "<Major, Minor>: <%d, %d>\n", MAJOR(first), MINOR(first));      return 0;  }    static void \_\_exit ofcd\_exit(void) /\* Destructor \*/  {      unregister\_chrdev\_region(first, 3);      printk(KERN\_INFO "Alvida: ofcd unregistered");  }    module\_init(ofcd\_init);  module\_exit(ofcd\_exit);    MODULE\_LICENSE("GPL");  MODULE\_AUTHOR("Anil Kumar Pugalia <email\_at\_sarika-pugs\_dot\_com>");  MODULE\_DESCRIPTION("Our First Character Driver"); |

Then, Shweta repeated the usual steps that she’d learnt for the first driver:

* Build the driver (.ko file) by running make.
* Load the driver using insmod.
* List the loaded modules using lsmod.
* Unload the driver using rmmod.

Summing up

Additionally, before unloading the driver, she peeped into the /proc/devices kernel window to look for the registered major number with the name “Shweta”, using cat /proc/devices. It was right there. However, she couldn’t find any device file created under /dev with the same major number, so she created them by hand, using mknod, and then tried reading and writing those. Figure 2 shows all these steps.

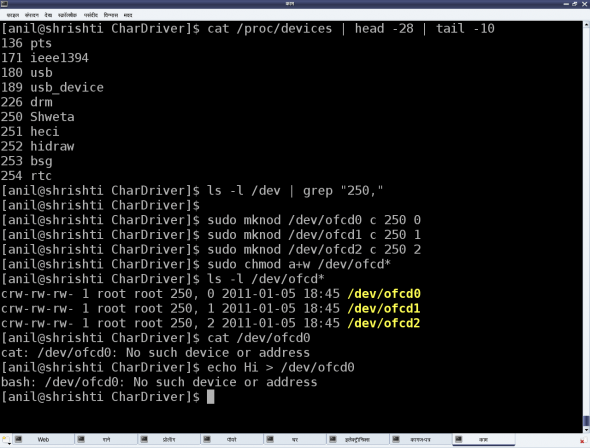
[](http://www.opensourceforu.com/wp-content/uploads/2011/02/figure_8_char_dev_file_experiments.png)

Figure 2: Character device file experiments

Please note that the major number 250 may vary from system to system, based on availability. Figure 2 also shows the results Shweta got from reading and writing one of the device files. That reminded her that the second step to connect the device file with the device driver — which is linking the device file operations to the device driver functions — was not yet done. She realised that she needed to dig around for more information to complete this step, and also to figure out the reason for the missing device files under /dev.

# Character Device Files — Creation & Operations

In my [previous article](http://www.opensourceforu.com/2011/02/linux-character-drivers/), I had mentioned that even with the registration for the <major, minor> device range, the device files were not created under /dev — instead, Shweta had to create them manually, using mknod. However, on further study, Shweta figured out a way to automatically create the device files, using the udev daemon. She also learnt the second step to connect the device file with the device driver — linking the device file operations to the device driver functions. Here is what she learnt.

## Automatic creation of device files

Earlier, in kernel 2.4, the automatic creation of device files was done by the kernel itself, by calling the appropriate APIs of devfs. However, as the kernel evolved, kernel developers realised that device files were more related to user-space and hence, as a policy, that is where they ought to be dealt with, not at the kernel. Based on this idea, the kernel now only populates the appropriate device class and device information into the /sys window, for the device under consideration. User-space then needs to interpret it and take appropriate action. In most Linux desktop systems, the udev daemon picks up that information, and accordingly creates the device files.

udev can be further configured via its configuration files to tune the device file names, their permissions, their types, etc. So, as far as the driver is concerned, the appropriate /sys entries need to be populated using the Linux device model APIs declared in <linux/device.h>. The rest should be handled by udev. The device class is created as follows:

|  |
| --- |
| struct class \*cl = class\_create(THIS\_MODULE, "<device class name>"); |

Then, the device info (<major, minor>) under this class is populated by:

|  |
| --- |
| device\_create(cl, NULL, first, NULL, "<device name format>", ...); |

Here, the first is dev\_t with the corresponding <major, minor>. The corresponding complementary or the inverse calls, which should be called in chronologically reverse order, are as follows:

|  |
| --- |
| device\_destroy(cl, first);  class\_destroy(cl); |

Refer to Figure 1 for the /sys entries created using chardrv as the <device class name> and mynull as the <device name format>. That also shows the device file, created by udev, based on the <major>:<minor> entry in the dev file.

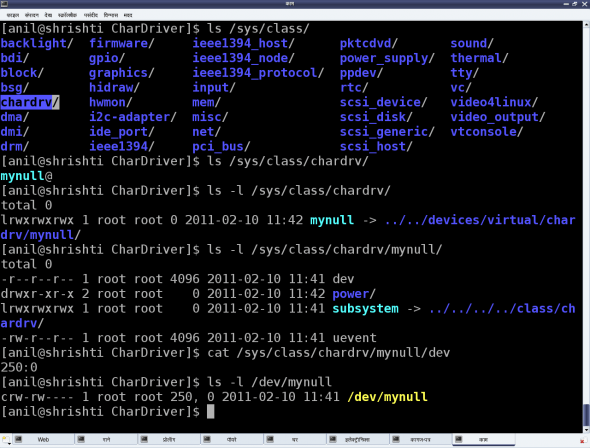
[](http://www.opensourceforu.com/wp-content/uploads/2011/04/figure_1_auto_dev_file_creation.png)

Figure 1: Automatic device file creation

In case of multiple minors, the device\_create() and device\_destroy() APIs may be put in the for loop, and the <device name format> string could be useful. For example, the device\_create() call in a for loop indexed by icould be as follows:

|  |
| --- |
| device\_create(cl, NULL, MKNOD(MAJOR(first), MINOR(first) + i), NULL, "mynull%d", i); |

## File operations

Whatever system calls (or, more commonly, file operations) we talk of on a regular file, are applicable to device files as well. That’s what we say: a file is a file, and in Linux, almost everything is a file from the user-space perspective. The difference lies in the kernel space, where the virtual file system (VFS) decodes the file type and transfers the file operations to the appropriate channel, like a filesystem module in case of a regular file or directory, and the corresponding device driver in case of a device file. Our discussion focuses on the second case.

Now, for VFS to pass the device file operations onto the driver, it should have been informed about it. And yes, that is what is called registering the file operations by the driver with the VFS. This involves two steps. (The parenthesised code refers to the “null driver” code below.)

First, let’s fill in a file operations structure (struct file\_operations pugs\_fops) with the desired file operations (my\_open, my\_close, my\_read, my\_write, …) and initialise the character device structure (struct cdev c\_dev) with that, using cdev\_init().

Then, hand this structure to the VFS using the call cdev\_add(). Both cdev\_init() and cdev\_add() are declared in <linux/cdev.h>. Obviously, the actual file operations (my\_open, my\_close, my\_read, my\_write) also had to be coded.

So, to start with, let’s keep them as simple as possible — let’s say, as easy as the “null driver”.

## The null driver

Following these steps, Shweta put the pieces together, attempting her first character device driver. Let’s see what the outcome was. Here’s the complete code — ofcd.c:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78 | #include <linux/module.h>  #include <linux/version.h>  #include <linux/kernel.h>  #include <linux/types.h>  #include <linux/kdev\_t.h>  #include <linux/fs.h>  #include <linux/device.h>  #include <linux/cdev.h>    static dev\_t first; // Global variable for the first device number  static struct cdev c\_dev; // Global variable for the character device structure  static struct class \*cl; // Global variable for the device class  static int my\_open(struct inode \*i, struct file \*f)  {    printk(KERN\_INFO "Driver: open()\n");    return 0;  }    static int my\_close(struct inode \*i, struct file \*f)  {    printk(KERN\_INFO "Driver: close()\n");    return 0;  }    static ssize\_t my\_read(struct file \*f, char \_\_user \*buf, size\_t    len, loff\_t \*off)  {    printk(KERN\_INFO "Driver: read()\n");    return 0;  }    static ssize\_t my\_write(struct file \*f, const char \_\_user \*buf,    size\_t len, loff\_t \*off)  {    printk(KERN\_INFO "Driver: write()\n");    return len;  }    static struct file\_operations pugs\_fops =  {    .owner = THIS\_MODULE,    .open = my\_open,    .release = my\_close,    .read = my\_read,    .write = my\_write  };    static int \_\_init ofcd\_init(void) /\* Constructor \*/  {    printk(KERN\_INFO "Namaskar: ofcd registered");    if (alloc\_chrdev\_region(&first, 0, 1, "Shweta") < 0)    {      return -1;    }      if ((cl = class\_create(THIS\_MODULE, "chardrv")) == NULL)    {      unregister\_chrdev\_region(first, 1);      return -1;    }      if (device\_create(cl, NULL, first, NULL, "mynull") == NULL)    {      class\_destroy(cl);      unregister\_chrdev\_region(first, 1);      return -1;    }      cdev\_init(&c\_dev, &pugs\_fops);      if (cdev\_add(&c\_dev, first, 1) == -1)    {      device\_destroy(cl, first);      class\_destroy(cl);      unregister\_chrdev\_region(first, 1);      return -1;    }    return 0;  }    static void \_\_exit ofcd\_exit(void) /\* Destructor \*/  {    cdev\_del(&c\_dev);    device\_destroy(cl, first);    class\_destroy(cl);    unregister\_chrdev\_region(first, 1);    printk(KERN\_INFO "Alvida: ofcd unregistered");  }    module\_init(ofcd\_init);  module\_exit(ofcd\_exit);  MODULE\_LICENSE("GPL");  MODULE\_AUTHOR("Anil Kumar Pugalia <email\_at\_sarika-pugs\_dot\_com>");  MODULE\_DESCRIPTION("Our First Character Driver"); |

Shweta repeated the usual build process, with some new test steps, as follows:

1. Build the driver (.ko file) by running make.
2. Load the driver using insmod.
3. List the loaded modules using lsmod.
4. List the major number allocated, using cat /proc/devices.
5. “null driver”-specific experiments (refer to Figure 2 for details).
6. Unload the driver using rmmod.

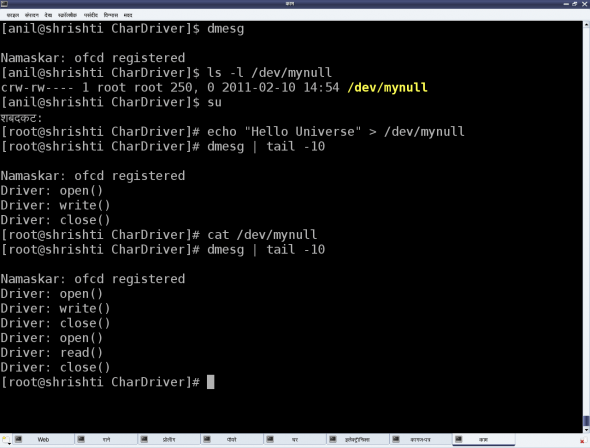
[](http://www.opensourceforu.com/wp-content/uploads/2011/04/figure_2_null_driver_experiments.png)

Figure 2: 'null driver' experiments

## Summing up

Shweta was certainly happy; all on her own, she’d got a character driver written, which works the same as the standard /dev/null device file. To understand what this means, check the <major, minor> tuple for /dev/null, and similarly, also try out the echo and cat commands with it.

However, one thing began to bother Shweta. She had got her own calls (my\_open, my\_close, my\_read, my\_write) in her driver, but wondered why they worked so unusually, unlike any regular file system calls. What was unusual? Whatever was written, she got nothing when reading — unusual, at least from the regular file operations’ perspective. How would she crack this problem? Watch out for the next article.

# Decoding Character Device File Operations

In our [previous article](http://www.opensourceforu.com/2011/04/character-device-files-creation-operations/), we saw how Shweta was puzzled by not being able to read any data, even after writing into the /dev/mynull character device file. Suddenly, a bell rang — not inside her head, but a real one at the door. And for sure, there was Pugs.

“How come you’re here?” exclaimed Shweta.

“I saw your tweet. It’s cool that you cracked your first character driver all on your own. That’s amazing. So, what are you up to now?” asked Pugs.

“I’ll tell you, on the condition that you do not play spoil sport,” replied Shweta.

Pugs smiled, “Okay, I’ll only give you advice.”

“And that too, only if I ask for it! I am trying to understand character device file operations,” said Shweta.

Pugs perked up, saying, “I have an idea. Why don’t you decode and then explain what you’ve understood about it?”

Shweta felt that was a good idea. She tail‘ed the dmesg log to observe the printk output from her driver. Alongside, she opened her null driver code on her console, specifically observing the device file operations my\_open, my\_close, my\_read, and my\_write.

|  |
| --- |
| static int my\_open(struct inode \*i, struct file \*f)  {      printk(KERN\_INFO "Driver: open()\n");      return 0;  }  static int my\_close(struct inode \*i, struct file \*f)  {      printk(KERN\_INFO "Driver: close()\n");      return 0;  }  static ssize\_t my\_read(struct file \*f, char \_\_user \*buf, size\_t len, loff\_t \*off)  {      printk(KERN\_INFO "Driver: read()\n");      return 0;  }  static ssize\_t my\_write(struct file \*f, const char \_\_user \*buf, size\_t len, loff\_t \*off)  {      printk(KERN\_INFO "Driver: write()\n");      return len;  } |

Based on the earlier understanding of the return value of the functions in the kernel, my\_open() and my\_close()are trivial, their return types being int, and both of them returning zero, means success.

However, the return types of both my\_read() and my\_write() are not int, rather, it is ssize\_t. On further digging through kernel headers, that turns out to be a signed word. So, returning a negative number would be a usual error. But a non-negative return value would have additional meaning. For the read operation, it would be the number of bytes read, and for the write operation, it would be the number of bytes written.

## Reading the device file

To understand this in detail, the complete flow has to be given a relook. Let’s take the read operation first. When the user does a read from the device file /dev/mynull, that system call comes to the virtual file system (VFS) layer in the kernel. VFS decodes the <major, minor> tuple, and figures out that it needs to redirect it to the driver’s function my\_read(), that’s registered with it. So from that angle, my\_read() is invoked as a request to read, from us — the device-driver writers. And hence, its return value would indicate to the requesters (i.e., the users), how many bytes they are getting from the read request.

In our null driver example, we returned zero — which meant no bytes available, or in other words, the end of the file. And hence, when the device file is being read, the result is always nothing, independent of what is written into it.

“Hmmm… So, if I change it to 1, would it start giving me some data?” asked Pugs, by way of verifying.

Shweta paused for a while, looked at the parameters of the function my\_read() and answered in the affirmative, but with a caveat — the data sent would be some junk data, since my\_read() is not really populating data into buf(the buffer variable that is the second parameter of my\_read(), provided by the user). In fact, my\_read() should write data into buf, according to len (the third parameter to the function), the count in bytes requested by the user.

To be more specific, it should write less than, or equal to, len bytes of data into buf, and the number of bytes written should be passed back as the return value. No, this is not a typo — in the read operation, device-driver writers “write” into the user-supplied buffer. We read the data from (possibly) an underlying device, and then write that data into the user buffer, so that the user can read it. “That’s really smart of you,” said Pugs, sarcastically.

## Writing into the device file

The write operation is the reverse. The user provides len (the third parameter of my\_write()) bytes of data to be written, in buf (the second parameter of my\_write()). The my\_write() function would read that data and possibly write it to an underlying device, and return the number of bytes that have been successfully written.

“Aha!! That’s why all my writes into /dev/ mynull have been successful, without actually doing any read or write,” exclaimed Shweta, filled with happiness at understanding the complete flow of device file operations.

## Preserving the last character

With Shweta not giving Pugs any chance to correct her, he came up with a challenge. “Okay. Seems like you are thoroughly clear with the read/write fundamentals; so, here’s a question for you. Can you modify these my\_read()and my\_write() functions such that whenever I read /dev/mynull, I get the last character written into /dev/mynull?”

Confidently, Shweta took on the challenge, and modified my\_read() and my\_write() as follows, adding a static global character variable:

|  |
| --- |
| static char c;    static ssize\_t my\_read(struct file \*f, char \_\_user \*buf, size\_t len, loff\_t \*off)  {      printk(KERN\_INFO "Driver: read()\n");      buf[0] = c;      return 1;  }  static ssize\_t my\_write(struct file \*f, const char \_\_user \*buf, size\_t len, loff\_t \*off)  {      printk(KERN\_INFO "Driver: write()\n");      c = buf[len – 1];      return len;  } |

“Almost there, but what if the user has provided an invalid buffer, or if the user buffer is swapped out. Wouldn’t this direct access of the user-space buf just crash and oops the kernel?” pounced Pugs.

Shweta, refusing to be intimidated, dived into her collated material and figured out that there are two APIs just to ensure that user-space buffers are safe to access, and then updated them. With the complete understanding of the APIs, she rewrote the above code snippet as follows:

|  |
| --- |
| static char c;    static ssize\_t my\_read(struct file \*f, char \_\_user \*buf, size\_t len, loff\_t \*off)  {      printk(KERN\_INFO "Driver: read()\n");      if (copy\_to\_user(buf, &c, 1) != 0)          return -EFAULT;      else          return 1;  }  static ssize\_t my\_write(struct file \*f, const char \_\_user \*buf, size\_t len, loff\_t \*off)  {      printk(KERN\_INFO "Driver: write()\n");      if (copy\_from\_user(&c, buf + len – 1, 1) != 0)          return -EFAULT;      else          return len;  } |

Then Shweta repeated the usual build-and-test steps as follows:

1. Build the modified “null” driver (.ko file) by running make.
2. Load the driver using insmod.
3. Write into /dev/mynull, say, using echo -n "Pugs" > /dev/ mynull
4. Read from /dev/mynull using cat /dev/mynull (stop by using Ctrl+C)
5. Unload the driver using rmmod.

On cat‘ing /dev/mynull, the output was a non-stop infinite sequence of s, as my\_read() gives the last one character forever. So, Pugs intervened and pressed Ctrl+C to stop the infinite read, and tried to explain, “If this is to be changed to ‘the last character only once’, my\_read() needs to return 1 the first time, and zero from the second time onwards. This can be achieved using off (the fourth parameter of my\_read()).”