



Using DEVFS

How device programmers can write code for the devfs environment

Alessandro Rubini

Independent consultant

August 2000

Copyright Linux Magazine ©2000

The role of the kernel is mostly related to hardware control, as user-space programs need a way of referring to hardware devices that they wish to use. Some hardware devices are used implicitly, through interfaces such as sockets or filesystems. However, it is often necessary to refer to a hardware device directly -- such as a particular serial port or hard-disk partition. This is accomplished through the use of special device files that are usually found in the `/dev` directory. This article introduces devfs.

Contents:

[Registering an entry point](#)

[Avoiding major and minor numbers](#)

[More than that](#)

[About the author](#)

The special files are not associated with data stored in the disk; rather, they correspond to particular hardware devices. When user programs access and use these special files, the operation is passed to a device driver in the system kernel. For example, when you issue an `open` and a `read` on the `/dev/ttyS0` file, data is read from a serial port; the serial-port device driver is invoked whenever `/dev/ttyS0` is accessed.

All UNIX systems provide a directory, conventionally called `/dev`, where device files are held; however, each UNIX variant uses a different name layout inside that directory. Until recently, Linux systems were required to store the `/dev` directory on disk. What this means is that `/dev` includes directory entries for every possible device driver, which may not correspond to the particular hardware available on the system.

The static nature of information stored on disk makes it difficult to dynamically add and remove device files whenever drivers are loaded and unloaded from the kernel, which makes it impossible to rearrange the layout of entries in `/dev` should that need arise. Another disadvantage of the disk-layout mechanism for `/dev` is that the driver hierarchy must be tied to the idea of major and minor device numbers, which are the means used to associate a device name to the device driver in charge of hardware operations. In Linux, major and minor numbers are currently limited to eight bits wide, and changing that figure is not trivial. This effectively limits the number of devices and different device drivers that can be used by the system.

Linux kernel version 2.3.46 introduced "device filesystem," or `devfs`, support in the official kernel tree. `devfs` provides a new filesystem type to be used for `/dev`. This filesystem keeps track of `/dev` layout and entries from within the kernel, without using on-disk storage. This means that new entries can appear in `/dev` as device drivers are loaded and new hardware is detected by the system. This facility is marked as experimental, and its use is expected to remain optional, as some environments (such as embedded systems) may still prefer to use the old approach.

In this article I'm going to give only a brief introduction to `devfs`, skipping over its setup and configuration. More detailed documentation on those issues is available elsewhere; one good source of information, for instance, is the file `Documentation/filesystems/devfs/README` found in newer kernel source trees.

I'll show how device programmers can write code that fits in with the `devfs` environment. The discussion and sample code here are based on version 2.2.14 of the kernel, which has been patched with `devfs-patch-v99.11.gz`, available from <http://ftp.atnf.csiro.au/pub/people/rgooch/linux>.

The sample module discussed here, called `drums` (short for "devfs Resources in User Module Sample") is available from <http://ftp.linux.it/pub/People/rubini>.

Registering an entry point

A device driver that wants to register its entry point within the `devfs` filesystem should call one of the forms of the `devfs_register` function. The `devfs` kernel interface is prototyped in the header file `<linux/devfs_fs_kernel.h>`. Let's imagine for example that we want to register a character device driver. In this case the function to call is:

```

devfs_handle_t devfs_register
    (devfs_handle_t dir,const char *name,
     unsigned int namelen,unsigned int
     flags,unsigned int major,unsigned
     int minor,umode_t mode,uid_t uid,
     gid_t gid,void *ops, void *info);

```

Given the huge list of arguments, the function can register pretty much anything and can assign the desired ownership and permissions to the file. The current version of *devfs* does not allow registration of directories and symbolic links using this function, but there are other functions that can create such files.

In a perfectly *devfs*-ized world, *devfs_register* would be everything that's needed to create a */dev* entry point for a device. However, you may want to allow the superuser to create a non-*devfs* entry point, using the conventional *mknod* command. In order to support this, any calls to *register_chrdev* or *register_blkdev* should be replaced by calls to *devfs_register_chrdev* and *devfs_register_blkdev*, respectively, which take the same arguments.

Both *devfs_register_chrdev* and *devfs_register_blkdev* are simple wrappers around *register_chrdev* and *register_blkdev*. They either call the old-style function or don't do anything, according to whether or not the command-line option of *devfs=only* has been passed to the kernel at boot time. If *devfs* is the only way to access devices, these functions don't do anything, so any device file created outside of *devfs* will not be associated to any device driver.

With this background, *Listing One* shows how *drums* registers its entry points, which include a */dev/drums* directory and a few files.

Listing One: Registering devfs entries

```

devfs_register_chrdev(DRUMS_MAJOR, "drums", &drums_fops);
drums_dir = devfs_mk_dir(NULL, "drums", 0, NULL);
for (i=0; i<DRUMS_NR_DEV; i++) {
    drums_devs[i]=devfs_register(drums_dir/* parent dir*/,
                                drums_strings[i], DRUMS_NAME_LEN,
                                DEVFS_FL_NONE, DRUMS_MAJOR, i/*minor*/,
                                S_IFCHR | S_IRUGO, 0, 0,
                                &drums_fops, NULL);
}

```

Once registered, the devices behave pretty much like any conventional device, and you can even *chown* and *chmod* them. When you read data from any of the *drums* special files, you'll get back the "note" associated to the specific device, repeated over and over (see *Listing Two*).

Listing Two: Drum data

```

borea.root# ls -l /dev/drums
total 0
cr--r--r--    1 root    root    60,    0 Jan1 1970    bam
cr--r--r--    1 root    root    60,    1 Jan1 1970    bum
cr--r--r--    1 root    root    60,    2 Jan1 1970    pam
cr--r--r--    1 root    root    60,    3 Jan1 1970    pum
cr--r--r--    1 root    root    60,    4 Jan1 1970    tam
cr--r--r--    1 root    root    60,    5 Jan1 1970    tum
borea.root# head -2 /dev/drums/bam
bam
bam
borea.root# head -100 /dev/drums/tum | uniq
tum

```

The implementation of *drums* is pretty standard: The minor number of the device being read is used to choose which string to return to user space; the string being returned is the same as *drums_strings[i]* used in registering the device name, as you can see in *Listing Three*.

Listing Three: Choosing which string to return

```

int minor = MINOR(inode->i_rdev);

if (count > DRUMS_TXT_LEN) count = DRUMS_TXT_LEN;
copy_to_user(buf, drums_strings[minor],count);

```

Unregistering the devices at unload time is easy. You just need to call `devfs_unregister` for each entry point you registered. Also, if you called `devfs_register_chrdev` you should now call `devfs_unregister_chrdev`. Unregistering is shown in *Listing Four*.

Listing Four: Unregistering devfs entries

```
for (i=0; i<DRUMS_NR_DEV; i++)
    devfs_unregister(drums_devs[i]);
devfs_unregister(drums_dir);
devfs_unregister_chrdev(DRUMS_MAJOR, "drums");
```

Working without major and minor numbers

If your device is meant to be available only via *devfs*, you can choose to avoid major and minor numbers altogether. This is because when a *devfs* node is opened, the kernel doesn't need to use the device numbers, as the driver has already provided the *file_operations* structure that must be used to act on that device.

To get automatic device numbers, the only thing that's needed is to specify `DEVFS_FL_AUTO_DEVNUM` in the *flags* argument to `devfs_register`. The major and minor arguments are then ignored, and the filesystem will automatically choose a major/minor pair for your device. What is most interesting in using automatic device numbers is that the device driver can no longer base its operation on the minor number, since it won't be known at compile time.

Rather, the driver must use the *private_data* field that is part of the file structure. Most drivers that use the field internally assign its value at open time based on the minor number being opened, and use it in the other device methods (*read*, *write*, etc.). With *devfs* you are allowed to choose your *private_data* pointer before the device is opened, and the chosen value can be passed to `devfs_register` as the last argument. This allows you to associate a particular data structure with each *devfs* entry point and use that to determine which entry point has been accessed within the device driver.

The historical role of the device numbers is eliminated by *devfs*: The major number is unneeded because each device declares its operations, and similarly, the minor number is not needed because each device declares its private information. The only remaining problem may be in some user-space program, which might expect the major number to be consistent across similar devices.

In the *drums* module, you'll find *tambourine* and *timpani* as examples of automatic assignment of device numbers. Their appearance in the system is shown in *Listing Five* and the code lines that implement them are shown below in *Listing Six*.

Listing Five: Tambourine and Timpani

```
borea.root# ls -l /devfs/timpani /devfs/tambourine
cr--r--r--  1 root    root   144,  3 Jan 1 1970  /devfs/tambourine
cr--r--r--  1 root    root   144,  4 Jan 1 1970  /devfs/timpani
borea.root# head -1 /devfs/timpani
boom
borea.root# head -1 /devfs/tambourine
rattle
```

Listing Six: Automatic major/minor number assignment

```
/* init_module: register tambourine and timpani */
drums_tambourine = devfs_register(NULL, "tambourine", 0,DEVFS_FL_AUTO_DEVNUM, 0, 0,
    S_IFCHR | S_IRUGO, 0, 0,&drums_fops, (void *)"rattle\n");
drums_timpani = devfs_register(NULL, "timpani", 0,DEVFS_FL_AUTO_DEVNUM, 0, 0,
    S_IFCHR | S_IRUGO, 0, 0,&drums_fops, (void *)"boom\n");

/* this is the read() implementation */
txt = filp->private_data;
if (count > strlen(txt)) count = strlen(txt);
copy_to_user(buf, txt, count);
*offp += count;
return count;
```

The ability to work without using device numbers directly is very important, because the Linux device space is not far from exhaustion, due to its sparse nature: A major number is assigned for every device driver, even though most systems have only a dozen or so drivers installed.

More than that

While the sample *drums* module shows only the basic functionality of *devfs*, the interface exported by `devfs_fs_kernel.h` offers much more. The filesystem can host conventional files, symbolic links, and everything that can live in a conventional filesystem.

The filesystem is currently marked as experimental, even though the current *devfs* implementation is pretty stable. The problem with *devfs* as I write this is that the actual use of its features must be somehow standardized to prevent possible cluttering of the *devfs* name space. As a matter of fact, kernel developers are still discussing the suitability of *procfs* and *devfs* for all system configuration, in order to find the best and cleanest way to access system configuration and resources.

While non-*devfs* systems use less memory, and a tiny conventional */dev* directory is currently still the best option for small embedded systems, the availability of *devfs* opens a new range of options for driver developers and simplifies users' lives in adding new device drivers to their systems, since now the driver module can do everything that is needed to grant user-space access to the hardware.

About the author

Alessandro Rubini is an independent consultant based in Italy. He writes uninteresting device drivers and uninteresting applications like GNU barcode, which gained him his preferred email address: rubini@gnu.org.

Copyright Linux Magazine ©2000
Reprinted with permission.

What do you think of this article?

Killer! (5)

Good stuff (4)

So-so; not bad (3)

Needs work (2)

Lame! (1)

Comments?

[Privacy](#)

[Legal](#)

[Contact](#)