# udev – A Userspace Implementation of devfs

*Greg Kroah-Hartman*[*]
IBM Corp.
*Linux Technology Center*
greg@kroah.com, gregkh@us.ibm.com

## Abstract

Starting with the 2.5 kernel, all physical and virtual devices in a system are visible to userspace in a hierarchal fashion through sysfs. /sbin/hotplug provides a notification to userspace when any device is added or removed from the system. Using these two features, a userspace implementation of a dynamic /dev is now possible that can provide a very flexible device naming policy.

This paper will discuss udev, a program that replaces the functionality of devfs (only providing /dev entries for devices that are in the system at any moment in time), and allows for features that were previously not able to be done through devfs alone, such as:

- Persistent naming for devices when they move around the device tree.
- Notification of external systems of device changes.
- A flexible device naming scheme.
- Allow the kernel to use dynamic major and minor numbers
- Move all naming policy out of the kernel.

This paper will describe why such a userspace program is superior to a kernel based devfs, and detail the design decisions that went into its creation. The paper will also describe how udev works, how to write plugins that extend the functionality of it (different naming schemes, etc.), and different trade offs that were made in order to provide a working system.

## 1 Introduction

The /dev directory on a Linux machine is where all of the device files for the system should be located.[2] A device file is how a user program can access a specific hardware device or function. For example, the device file /dev/hda is traditionally used to represent the first IDE drive in the system. The name hda corresponds to both a major and a minor number, which is used by the kernel to determine what hardware device to talk to. Currently a very wide range of names that match up to different major and minor numbers have been defined.

All major and minor numbers are assigned a name that matches up with a type of device. This allocation is done by The Linux Assigned Names And Numbers Authority (LANANA)[4] and the current device list can be always be found on their web site at http://www.lanana.org/docs/device-list/devices.txt

[*]This work represents the view of the author and does not necessarily represent the view of IBM.

As Linux gains support for new kinds of devices, they need to be assigned a major and minor number range in order for the user to be able to access them through the `/dev` directory (one alternative to this is to provide access through a filesystem [3]). In the kernel versions 2.4 and earlier, the valid range of major numbers was 1-255 and minor numbers was 1-255. Because of this limited range, a freeze was placed on allocating new major and minor numbers during the 2.3 development cycle. This freeze has since been lifted, and the 2.6 kernel should see an increase in the range of major and minor numbers available for use.

## 2 Problems with current scheme

### 2.1 What `/dev` entry is which device

When the kernel finds a new piece of hardware, it typically assigns the next major/minor pair for that kind of hardware to the device. So, on boot, the first USB printer found would be assigned the major number 180 and minor number 0 which is referenced in `/dev` as `/dev/usb/lp0`. The second USB printer would be assigned major number 180 and minor number 1 which is referenced in `/dev` as `/dev/usb/lp1`. If the user rearranges the USB topology, perhaps adding a USB hub in order to support more USB devices in the system, the USB probing order of the printers might change the next time the computer is booted, reversing the assignment of the different minor number to the two printers.

This same situation holds true for almost any kind of device that can be removed or added while the computer is powered up. With the advent of PCI hotplug enabled systems, and hot-pluggable busses like IEEE1394, USB, and CardBus, almost all devices have this problem.

With the advent of the `sysfs` filesystem in the 2.5 kernel, the problem of determining which device minor is assigned to which physical device is now much easier to determine. For a system with two different USB printers plugged into it, the `sysfs` `/sys/class/usb` directory tree can look like Figure 1. Within the individual USB device directories pointed to by the `lp0/device` and `lp1/device` symbolic links, a lot of USB specific information can be determined, such as the manufacturer of the device, and the (hopefully unique) serial number.

As can be seen by the serial files in Figure 1, the `/dev/usb/lp0` device file is associated with the USB printer with serial number `HXOLL0012202323480`, and the `/dev/usb/lp1` device file is associated with the USB printer with serial number `W09090207101241330`.

If these printers are moved around, by placing them both behind a USB hub, they might get renamed, as they are probed in a different order on startup.

In Figure 2, `/dev/usb/lp0` is assigned to the USB printer with the serial number `W09090207101241330` due to this different probing order.

`sysfs` now enables a user to determine which device has been assigned by the kernel to which device file. This is a very powerful association that has not been previously easily available. However, a user generally does not care that `/dev/usb/lp0` and `/dev/usb/lp1` are now reversed and should be changed in some configuration file somewhere, they just want to always be able to print to the proper printer, no matter where it is in the USB device tree.

```
/sys/class/usb/
|-- lp0
|   |-- dev
|   |-- device -> ../../../devices/pci0/00:09.0/usb1/1-1/1-1:0
|   `-- driver -> ../../../bus/usb/drivers/usblp
`-- lp1
    |-- dev
    |-- device -> ../../../devices/pci0/00:0d.0/usb3/3-1/3-1:0
    `-- driver -> ../../../bus/usb/drivers/usblp

$ cat /sys/class/usb/lp0/device/serial
HXOLL0012202323480
$ cat /sys/class/usb/lp1/device/serial
W09090207101241330
```

Figure 1: Two USB printers plugged into different USB busses

```
$ tree /sys/class/usb/
/sys/class/usb/
|-- lp0
|   |-- dev
|   |-- device -> ../../../devices/pci0/00:09.0/usb1/1-1/1-1.1/1-1.1:0
|   `-- driver -> ../../../bus/usb/drivers/usblp
`-- lp1
    |-- dev
    |-- device -> ../../../devices/pci0/00:09.0/usb1/1-1/1-1.4/1-1.4:0
    `-- driver -> ../../../bus/usb/drivers/usblp

$ cat /sys/class/usb/lp0/device/serial
W09090207101241330
$ cat /sys/class/usb/lp1/device/serial
HXOLL0012202323480
```

Figure 2: Same USB printers plugged into a USB hub

### 2.2 Not enough numbers

The current range of allowed major and minor numbers is 8 bits (0-255). Currently there are very few major numbers left for new character devices, and about half the number of major numbers available for block devices (block and character devices can use the same numbers, but the kernel treats them separately, giving the whole range for both types of devices.) This seems like a lot of free numbers, but there are users who want to use a very large number of disks all at the same time, for which the 8 bit scheme is too small. A common goal of some companies is to connect about 4,000 disks to a single system. For every disk, the kernel reserves 16 minor numbers, due to the possibility of there being up to 16 partitions on every disk. So 4,000 disks would require 64,000 different device files, needing at least 250 major num-

bers to handle all of them. This would work in the 8 bit scheme, as they all would fit, but, the majority of the current major numbers are already reserved. The disk subsystem can not just steal major numbers from other subsystems very easily. And if it did so, userspace would have to know which previously reserved major numbers are now being used by the SCSI disk subsystem.

Because of this limitation, a lot of people are pushing for an increase in the size of the major and minor number range, which would be required to be able to support more than 4,000 disks (some users talk of connecting 10,000 disks at once.) It looks like this work will go into the 2.6 kernel; however, the large problem remains of how to notify userspace which major and minor numbers are being used.

Even if the major number range is increased, the requirement of reserving major number ranges for different types of subsystems is still present. It still requires an external naming authority, and possibly we could run out of ranges sometime in the far future. If the kernel were to switch to a dynamic method of allocating major and minor numbers, for only the devices that were currently connected to the system, this authority would no longer be needed, and the range of numbers would never run out. The biggest problem with dynamic allocation is again, userspace has no idea which devices are assigned to which major and minor numbers.

A few kernel subsystems currently do allocate minor numbers dynamically. The USB to Serial subsystem has been doing this since the 2.2 kernel series, with great success. The biggest problem still is the user does not know what device is assigned to what number, and has to rely on looking in a kernel log to make that determination. With `sysfs` in the 2.5 kernel, this is much easier.

### 2.3 `/dev` is too big

Not all device files in the `/dev` directory of most distributions match up to a physical device that is currently connected to the computer at that time. Instead, the `/dev` directory is created when the operating system is initialized on the machine, populating the `/dev` directory with all known possible names. On a machine running Red Hat 9, the `/dev` directory holds over 18 thousand different entries. This large number of different entries soon becomes very unwieldy for users to try to determine exactly what devices are currently present.

Because of the large numbers of device files in the `/dev` directory, a number of operating systems have moved over to having the kernel itself manage the `/dev` directory, as the kernel always knows exactly what devices are present in the system. It does this by creating a ram based filesystem called `devfs`. Linux also has this option, and it has become popular over time with a number of different distributions (the Gentoo distribution being one of the more notable ones.)

### 2.4 `devfs`

A number of other Unix-like operating systems have solved a lot of the previously mentioned problems by using a kernel-based `devfs` filesystem. Linux also has a `devfs` filesystem, and for a number of people, this solves their immediate needs. However, the Linux-based `devfs` implementation still has a number of problems unsolved.

`devfs` does only show exactly what devices are currently in the system at any point in time, solving the "`/dev` is too big" issue. However, the names used by `devfs` are not the names that the LANANA authority has issued. Because of this, switching between a `devfs` system, and a static `/dev` system is a bit dif-

ficult, due to the number of different configuration files that need to be modified. The `devfs` authors have tried to address this problem and have provided some compatibility layers to emulate the `/dev` names.

Even with `devfs` running in compatibility mode, the Linux kernel is imposing a set naming policy on userspace. It is saying that the first IDE drive is going to be called `/dev/hda` or `/dev/ide/hd/c0b0t0u0` and there is nothing that a user can do about this. Generally, the Linux kernel developers do not like forcing any policies on userspace, when it can be helped. This naming policy should be moved out of the kernel, so that the kernel driver developers can focus not on naming arguments (of which the `devfs` naming arguments consumed many man years of time). In short, the kernel should not care what a user wants to call a device, but if `devfs` is used, this is not possible.

`devfs` also does not allow devices to be bound to major and minor numbers dynamically. The current `devfs` implementation still uses the same major and minor numbers that are assigned by LANANA. `devfs` can be modified to do dynamic allocation; however, no one has done so yet.

`devfs` also forces all of the device names and the naming database into kernel memory. Kernel memory can not be swapped out, and is always resident. For very large amounts of devices (like the previously mentioned 4,000 disks), the overhead of keeping all of the device names in kernel memory is not unsubstantial. During some testing of a wider major number range, one developer ran into memory starvation issues on a 32 bit Intel processor, just with a static `/dev` system. Add the overhead of 4,000 different disk names and structures to manage those names, and even less memory would be available for user programs to use.

## 3 `udev`'s goals

So, in light of all of the previously mentioned problems, the `udev` project was started. Its goals are the following:

- Run in userspace
- Create a dynamic `/dev`.
- Provide consistent device naming, if wanted.
- Provide a userspace API to access info about current system devices.

The first item, "run in userspace," is easily done by harnessing the fact that `/sbin/hotplug` generates an event for every device that is added or removed from the system, combined with the ability of `sysfs` to show all needed information about all devices.

The rest of the goals enable the `udev` project to be split into three separate subsystems:

1. `namedev` – handles all device naming
2. `libsysfs` – a standard library for accessing device information on the system.
3. `udev` – dynamic replacement for `/dev`

### 3.1 `namedev`

Due to the need for different naming schemes for devices, the device naming portion of udev has been moved into its own subsystem. This was done to move the naming policy decision out of the udev binary, allowing pluggable naming schemes to be developed by different groups. This device naming subsystem, namedev, presents a standard interface that udev can call to name a specific device.

With the initial releases of udev, the namedev logic is still provided in a few source files that get linked into the udev binary. There is currently only one naming scheme implemented, the one specified by LANANA[4]. This scheme is quite simple, as generally the sysfs representation of the device uses the same name, and will be suitable for the majority of current Linux users.

As the current kernel devfs provides a competing naming schema from LANANA, there has been some interest in providing a module that contains this, but this is currently unavailable due to lack of interest by the primary developers.

Part of the goal for the udev project is to provide a way for users to name devices based on a set of policies. The current version of namedev provides the user with a five step sequence for determining the name of a given device. These steps are consulted in order, and if the device's name can be determined at any step, that name is used. The existing steps are as follows:

1. label or serial number
2. bus device number
3. topology on bus
4. replace name
5. kernel name

In the first step, the device that is added to the system is checked to see if it has a unique identifier, based on that type of device. For example, on USB devices, the USB serial number is checked; for SCSI devices, the UUID is checked; for block devices, the filesystem label is checked. If this matches a identifier provided by the user (in a configuration file), the resulting name (again specified in the configuration file) is used.

The second step checks on the device's bus number. For a lot of busses, this number generally does not change over time, and all bus numbers are guaranteed to be unique at any one point in time in the system. A good example of this is PCI bus numbers, which rarely change on the majority of systems (however, BIOS upgrades, or hotplug PCI controllers, can renumber the PCI bus number the next time the machine is booted.) Again, if the bus number matches an identifier provided by the user, the resulting name is assigned to the device.

The third step checks the position of the device on the bus. For example, a USB device can be described as residing in the 3rd hub port of the hub plugged into the first port on the root hub. This topology will not change, unless the user physically moves the devices around, and is independent of any bus numbering changes that might occur between reboots of a machine. If the topology position on the bus matches the position provided by the user, the requested name is assigned to the device.

The fourth step is a simple string replacement. If the kernel name for a device matches the name specified here, the requested new name will be used in its place. This is useful for devices that users always know will have the same kernel name, but wish to name something different.

The fifth step is the catch-all step. If none of the previous steps have provided a name for this device, the default kernel name will be used for this device. For the majority of devices in a system, this is the rule that will be used, as it matches the way devices are named on a Linux system without devfs or udev.

Figure 3 shows an example namedev configuration file. This configuration file shows how the four different ways of overriding the default kernel naming scheme can be changed. The first two entries show how to specify a se-

```
# USB Epson printer to be called lp_epson
LABEL, BUS="usb", serial="HXOLL0012202323480", NAME="lp_epson"

# USB HP printer to be called lp_hp,
LABEL, BUS="usb", serial="W09090207101241330", NAME="lp_hp"

# sound card with PCI bus id 00:0b.0 to be the first sound card
NUMBER, BUS="pci", id="00:0b.0", NAME="dsp"

# sound card with PCI bus id 00:07.1 to be the second sound card
NUMBER, BUS="pci", id="00:07.1", NAME="dsp1"

# USB mouse plugged into the third port of the first hub to be
# called mouse0
TOPOLOGY, BUS="usb", place="1.3", NAME="mouse0"

# USB tablet plugged into the second port of the second hub to be
# called mouse1
TOPOLOGY, BUS="usb", place="2.2", NAME="mouse1"

# ttyUSB1 should always be called visor
REPLACE, KERNEL="ttyUSB1", NAME="visor"
```

Figure 3: Example `namedev` configuration file

rial number of a device to control what that device should be named. The third and fourth entries show how to override the bus probing order and name a device based on the specific bus id. The fifth and sixth entries show how the USB topology can be used to specify a device name, and the seventh entry shows how to do a simple name substitution.

### 3.2 `libsysfs`

There is a need for a common API to access device information in `sysfs` by a number of varied programs, not just the `udev` project. The device naming subsystem and the `udev` subsystem need to query a wide range of device information from a `sysfs` represented device. Instead of duplicating this logic around in different projects, splitting this logic of `sysfs` calls into a separate library that will sit on top of `sysfs` makes more sense. `sysfs` representations of different devices are not standard (PCI devices have different attributes from USB devices, etc.) so this is another reason for creating a common and standard library interface for querying device information.

Right now the current `udev` codebase is using an initial version of `libsysfs`, and the `libsysfs` codebase is under active development.

### 3.3 `udev`

The `udev` program will be responsible for talking to both the `namedev` and `libsysfs` libraries to accomplish the device naming policy that has been specified. The `udev` program is run whenever `/sbin/hotplug` is called by the kernel. It does this by adding a symlink to itself in the `/etc/hotplug.d/default` directory, which is searched by the `/sbin/hotplug` multiplexer script.

The `/sbin/hotplug` invocation by the ker-

nel exports a lot of device specific information on what action just happened (add or remove), what device type the action took place for (USB, PCI, etc.), and what device in the `sysfs` tree did the action. `udev` takes this information, calls `namedev` to determine the name it should give for this device (or the name that has already been given to this device if it is a remove event). If this is a new device that has been added, `udev` uses `libsysfs` to determine the major and minor number that should be used for the device file for this device, and then creates the device file in the `/dev` directory with the proper name and major/minor number. If this is a device that has been removed, then the device file in the `/dev` directory that had previously been created for this device will be removed.

## 4 Enhancements

There are a number of different enhancements that different users have asked for, that can be added to the existing `udev` implementation.

A lot of userspace programs want to be notified when a new device has been added or removed from the system. Gnome and KDE both want to add a new icon if a disk has been added, or possibly launch a sync program if a USB Palm device has been attached. The D-BUS project[1] has been created to help provide a simple way for applications to talk to one another using messages. It has been proposed that the `udev` program create a D-BUS message after it has created or removed a device file, so that any listening applications can act upon this event.

Currently, `namedev` uses a very simple configuration file, creating a simple ram based database that it uses to store all current device information, and device naming rules. It has been proposed that this database (if it can even really be called such a thing), be moved to a real, backing-store type database, in order to store a persistent view of the system, or to provide a more complex naming scheme for `udev` to use.

## 5 Thanks

The author would like to thank Daniel Stekloff of IBM who has helped shape the design of `udev` in many ways. Without his perseverance, `udev` might not even be working. He also provided the initial design documents for how `udev` could be split up into different pieces, allowing pluggable naming schemes and has been instrumental in the development of `libsysfs`. Also, without Pat Mochel's `sysfs` and driver model core, `udev` would not even have been possible to implement. The author is indebted to him for undertaking what most thought as an impossible task, and for allowing others to easily build on his common framework, allowing all users to see the "web woven by a spider on drugs"[5] that the kernel keeps track of.

## 6 Legal Statement

## References

[1] D-BUS project. `http://www.freedesktop.org/software/dbus/`.

[2] Linux Filesystem Hierarchy Standard. `http://www.pathname.com/fhs/2.2/`.

[3] Greg Kroah-Hartman. Putting a filesystem into a device driver. In *Linux.conf.au*, Perth, Australia, January 2003.

[4] The Linux Assigned Names And Numbers Authority. `http://www.lanana.org/`.

[5] Linux Weekly News. `http://lwn.net/Articles/31185/`.

# Proceedings of the Linux Symposium

July 23th–26th, 2003
Ottawa, Ontario
Canada

## Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

## Review Committee

Alan Cox, *Red Hat, Inc.*
Andi Kleen, *SuSE, GmbH*
Matthew Wilcox, *Hewlett-Packard*
Gerrit Huizenga, *IBM*
Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Martin K. Petersen, *Wild Open Source, Inc.*

## Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*