

Dynamic device management in Udev

DEVICE MANAGER

After three years of hanging around on the sidelines, Udev has finally ousted the legacy Dev-FS system. We take a look under the hood at the Udev device management system inside your Linux system.

BY RENE REBE, OLIVER FROMMEL, JENS-CHRISTOPH BRENDEL

Linux inherited the classic Unix adage “Everything is a file.” This abstraction has allowed programs to use device nodes (device files) to access computer hardware as if they were accessing an ordinary file. These special device files, which are opened, read, written to, or closed using the same system calls as text files, are distinguished by their names, types (block or character

device), major numbers, and minor numbers.

Previously Too Static

Device files are created by the *mknod* command during the installation phase, assuming legacy management. This approach creates a permanent entry in */dev* for each device the admin might connect to the system at any time in the future,

and this can easily mean thousands of entries.

This huge mess of entries, most of which are useless for the current system because they refer to devices that don't exist, makes it difficult to keep track of the configuration. For example, the directory and file structure does not tell you which devices exist or which have been correctly identified by drivers. Ad-

Hotplug Details

To allow dynamic device management to work, the Linux kernel has to have the *CONFIG_HOTPLUG* option built in; standard distributions build in the *CONFIG_HOTPLUG* option by default.

When a device driver adds or removes what is called a *Kobject* from the system, the kernel sends messages with notice of this change to the Udev daemon, or it runs the program residing at */proc/sys/kernel/hotplug* by calling the *kset_hotplug* function (the source code is in *lib/kobject.c*).

This program used to be at */sbin/hotplug*, and then it changed to */sbin/udevsend*, before being dropped in recent versions. The subsystem class is passed in as an argument.

The environmental variables provide more details. For example, *ACTION* supports values of *add* or *remove*. The *SEQNUM* variable is incremented for each call. *DEVPATH* specifies where in Sys-FS the device information is located, such as under */devices/pci0001:01/0001:01:*

19.0/usb2/2-1/2-1:1.0. And there are other parameters, such as *MAJOR*, *MINOR*, and *UDEV_EVENT*. Depending on the object type, additional variables may be exported: the vendor and product IDs for physical hardware, for example, or for USB *DEVICE* types, the corresponding USB-FS entry; this would be */proc/bus/usb/002/010* in the same example. Udev itself (formerly the hotplug scripts) assigns and loads the corresponding kernel module based on the hardware IDs in */lib/modules/\$ver*.

```

udevmonitor prints the received event from the kernel [EVENT]
and the event which udev sends out after rule processing [DEV]

EVENT [1145978268] add#/devices/pci0000:00/0000:00:02.2/usb/l1-5
EVENT [1145978268] add#/devices/pci0000:00/0000:00:02.2/usb/l1-5/l1-5:1.0
EVENT [1145978268] add#/class/dvb/dvb0.demux0
EVENT [1145978268] add#/class/dvb/dvb0.dvr0
EVENT [1145978268] add#/class/dvb/dvb0.net0
EVENT [1145978268] add#/class/dvb/dvb0.frontend0
EVENT [1145978268] add#/class/input/input3
EVENT [1145978268] add#/class/input/input3/event2
EVENT [1145978268] add#/class/usb_device/usbdev1.4
DEV [1145978268] add#/devices/pci0000:00/0000:00:02.2/usb/l1-5
DEV [1145978268] add#/class/dvb/dvb0.demux0
DEV [1145978268] add#/class/dvb/dvb0.dvr0
DEV [1145978268] add#/class/dvb/dvb0.net0
DEV [1145978268] add#/devices/pci0000:00/0000:00:02.2/usb/l1-5/l1-5:1.0
DEV [1145978268] add#/class/dvb/dvb0.frontend0
DEV [1145978270] add#/class/usb_device/usbdev1.4
DEV [1145978272] add#/class/input/input3
DEV [1145978272] add#/class/input/input3/event2

```

Figure 1: Udev actions during hot-plugging, recorded using udevmonitor.

ditionally, in this traditional approach, the order in which the user connects the devices determines which device file the kernel assigns to which device. The first SCSI disk detected is always mapped to the device file `/dev/sda`, for example; the next disk is mapped to `/dev/sdb` and so on. This approach can mean that the same device is accessible via different device files at different times, and the computer mounts it in different directories as a result.

An approach that creates and deletes device files on the fly when a user adds or removes hardware has proved more useful. In this case, name-to-device mappings are not based on the order the devices are detected but on a system of predefined rules. This means you can guarantee that a specific hard disk will always be available via the same file, and thus the disk will always be mounted at the same position in the filesystem, no matter what other devices share the bus.

For a while, the Dev-FS and Udev systems were competitors for the role of an intelligent device manager, but now Udev is the clear winner. Dev-FS was recently removed from the kernel, leaving Udev as the king of the hill [1]. Udev relies on the kernel hotplug mechanism (see the *Hotplug Details* box) to create device files in userspace.

When a device is connected or removed, the (ISA, USB, or PCI) bus controller signals the event with an interrupt. In response to the interrupt, the kernel ascertains the details of the newly-added device, either by using a

specific protocol to query the hardware controller or by searching at hard-coded addresses – such as the PCI card configuration area – for information stored there, more specifically, for the vendor and product IDs. The kernel then goes on to create a KObject for each new device; data on the device type (char or block) and the major and minor numbers are passed to and stored by the KObject via the global kernel component namespace introduced in kernel 2.6. The kernel then makes this information available via Sys-FS, which replaces the legacy proc filesystem. Sys-FS is normally mounted after `/sys`.

For older Udev versions, the kernel

informs user mode programs of the new device status by calling the program that resides at `/proc/sys/kernel/hotplug`, typically `/sbin/hotplug`. This program loads and configures any required drivers; the IDs referred to earlier tell the program which drivers to load. Modern Udev versions talk directly to the kernel and call external programs to perform the same actions directly.

Udev uses this approach both when devices are added to a running system (hot plugging) and for initialization during the boot phase (cold plugging). The kernel searches the buses for devices and creates *uevent* files based on the results. The results are stored in the virtual Sys-FS as the system does not have a writable root filesystem at this early stage. Later on, Udev generates events for the devices found at boot time, based on the results, just as if they had been hot-plugged. Figure 1 shows the procedure for plugging a USB DVB receiver, recorded using the *udevmonitor* diagnostic tool.

Up to Udev version 0.58, the hotplug package loads drivers and firmware. This approach, which applies to distros such as on the Suse 9.x versions, Fedora Core 4, Ubuntu Breezy Badger, and Debian Sarge, keeps the `/sbin/hotplug` hotplug manager, which acts as a multiplexer calling all the programs registered in `/etc/hotplug.d`. This should work fine, assuming a standard installation of the

Abstract Hardware

Device management is also the domain of another component, the Hardware Abstraction Layer (HAL). In addition to kernel and Udev information, HAL manages details of devices in XML-formatted FDI files (Device Information Files). Listing 1 shows an excerpt from an FDI file for a digital camera.

On Fedora, hotplug events are passed to the HAL daemon by the Udev subsystem using a rule located in `/etc/udev/rules.d/90-hal.rules`:

```
RUN+= "socket:/org/freedesktop/
hal/udev_event"
```

The *RUN+* element in this rule stipulates that this rule should be run for every other rule. communication is handled by a socket, as described previously.

The GNOME Network Manager [3] is a good example for the way all these com-

ponents interact. GNOME Network Manager uses the HAL daemon to monitor the network subsystem; HAL notifies the Network Manager via the D-Bus when changes occur, for example when wireless USB sticks are plugged or unplugged. Besides physical devices, HAL can also handle filesystems and ascertain the filesystem type, even for LUKS-encrypted partitions [4].

HAL handles the lion's share of hardware management on GNOME today, especially for hot-pluggable devices. To allow this to happen, the *gnome-volume-manager* process runs in the background. (GNOME users can run the *gnome-volume-properties* front-end to configure the manager.) There is even a front-end for HAL itself; it outputs a tree view of all connected devices (Figure 2). The *hal-device-manager* for Fedora is located in the *hal-gnome* package.

Table 1: Udev Variants

Version/Distribution	Fedora Core 5	SLES 9	SLES 10	Suse 10.0
Udev Version	udev-084-15	udev-021-36.49	udev-085-30.5	udev-068git20050831-9
HAL Version	hal-0.5.7-3	–	hal-0.5.6-33	hal-0.5.4-6
Hotplug script	none	/sbin/hotplug	none	none
Gnome Volume Manager	1.5.15-1	–	1.5.15-26.1	–
KDE Kioslave Media	yes	–	yes	yes
Ivman	–	–	ivman-0.6.9-16.3	–

hotplug and Udev packages. Starting with version 0.59, Udev handles both tasks. This approach is used in Fedora Core 5, Suse 10, and Ubuntu “Dapper Drake” (see also Table 1).

A Question of Preferences

The Udev configuration files are located in `/etc/udev`, or in `/lib/udev` with some distributions. The most important variables are stored in `/etc/udev.conf`, for example, the root directory for the device files, the path to the internal Udev database, and the directory with rules for creating and naming device files:

```
udev_root="/dev/"
udev_db="/dev/.udevdb"
udev_rules="/etc/udev/rules.d"
udev_log="err"
```

Udev provides two configuration points: `/etc/udev/rules.d/` contains files that govern device node naming and `/etc/udev/makedev.d/` contains scripts with the names of the static device files, which

are required for devices such as the legacy parallel port. Permissions no longer reside under `/etc/udev/permissions.d` for current Udev versions but are part of the normal Udev configuration.

Rule Syntax

The first part of every Udev rule specifies the condition that has to be fulfilled for Udev to execute or apply the second part. In a simple case, this condition can refer to the internal kernel name of a device. For example, the condition for a keyboard is `KERNEL = "kbd"`.

Conditions are indicated by double equals signs, just like in a programming language. More conditions can follow in a comma-separated list. Actions are introduced by a single equals sign. For example, `MODE = "0660"` sets the permissions. In a similar way, `OWNER` will set the owner, and `GROUP` will set the group. The `NAME` keyword specifies the device name, and wildcards are supported. For example, `%k` represents the kernel name discussed previously. A rule

that follows this pattern might look like this: `KERNEL = "isd*"`, `NAME = "%k"`, `MODE = "0660"`.

Udev will normally parse any matching rules until it runs out of rules. To cancel rule processing when a match occurs, specify `last_rule` below `OPTIONS`.

Script Scripts

Udev rules can call external programs, which are also evaluated as conditions. Udev will not perform the action unless the call leads to the required result. The manpage has an example for IDE CD ROMs that checks if a `/proc` directory exists to identify the device as a CD ROM:

```
KERNEL=="hd[a-z]",
PROGRAM=
"/bin/cat/proc/ide/%k/media",
RESULT="cdrom",
NAME="%k",
SYMLINK="cdrom%e"
```

Listing 1: FDI File for a Digital Camera

```
01 <deviceinfo version="0.2">
02   <device>
03     <match key="info.bus" string="usb">
04       <match key="usb.interface.class" int="0x06">
05         <match key="usb.interface.subclass" int="0x01">
06           <match key="usb.interface.protocol" int="0x01">
07             <merge key="info.category" type="string">camera</merge>
08             <append key="info.capabilities" type="strlist">camera</
append>
09             <merge key="camera.access_method" type="string">ptp</
merge>
10           </match>
11         </match>
12       </match>
13     </match>
14   </device>
15 </deviceinfo>
```

Listing 2: 95userpolicy

```
01 <?xml version="1.0"
encoding="ISO-8859-1"?> <!-- -
*- SGML *- -->
02 <!-- This .fdi file prevent
automount for every media
(storage devices)
03 e.g. floppy, CD/DVD, USB-
Stick, USB-Disk, external
harddisk. -->
04 <deviceinfo version="0.2">
05   <device>
06     <match key="storage.
policy.should_mount"
bool="true">
07       <merge key="storage.
policy.should_mount"
type="bool">>false</
merge>
08     </match>
09   </device>
10 </deviceinfo>
```


outputs errors only, *info* makes Udevd more talkative, and *debug* makes it really verbose. You can enter these values in the configuration file or use a utility to set the values at runtime: *udevcontrol log_priority = debug*. The *reload_rules* Udev control command tells the Udev daemon to reparse the modified rules.

Conclusions

Udev is a complex system that comprises multiple contributing layers: a daemon, hotplug scripts, and tools for event handling. The Udev system supports dynamic device management on modern Linux systems to reflect the demands placed on today's hardware. Udev takes mutual dependencies into consideration,

such as the dependencies that exist between partitions and hard disks, and manages today's ubiquitous hot-pluggable devices.

Although most distributions now use Udev, the package is still under development. This means that programs may be added and others discarded (including the former *udev* binary itself). The differences between the distributions, and even between different versions of the same distribution, are vast in some cases. And this means that most of the documentation on the Internet is obsolete.

If you use the Udev version provided by your distribution, you should not experience any difficulty. But if you intend

to modify the configuration yourself, you will have no alternative but to read the scripts provided with the package. You will find a few notes on programming your own Udev rules at [2], but these pages are not accurate in all respects. ■

INFO

- [1] Udev homepage: <http://www.kernel.org/pub/linux/utils/kernel/hotplug/udev.html>
- [2] Writing Udev Rules: http://reactivated.net/writing_udev_rules.html
- [3] Gnome Network Manager: <http://www.gnome.org/projects/NetworkManager>
- [4] LUKS: <http://www.redhat.com/magazine/012oct05/features/hal>

D-Bus

D-Bus is a communications system that allows desktop applications to interact among themselves and with the underlying layers down to the kernel and hardware layers. The D-Bus system is thus an Interprocess Communication (IPC) service, and it provides the infrastructure that helps applications talk both to each other and to parts of the operating system. Although there are Unix IPC mechanisms, they are restricted to signals, pipes, and the like.

The D-Bus model may sound familiar, as competitive approaches have been around for quite some time: Corba, Microsofts, DCOM, or hundreds of other projects, for example. Both KDE and Gnome originally experimented with their own Corba implementations. KDE introduced its proprietary DCOP system quite awhile back, whereas Gnome still has some legacy Corba in its Bonobo component system.

No matter what your personal stance on Corba may be, most developers who simply want to program a desktop application are overwhelmed by it. Even the simplest of Gnome applets requires expert knowledge of the complex component architecture. And this is why Bonobo has been on the Gnome sidelines for so long.

The idea was to make D-Bus more simple, and to give it a smaller footprint. The underlying Libdbus library only provides functions that support communication between two applications. Application developers do not normally resort to the library, preferring the Glib API-based Libdbus-Glib, which provides an object-oriented C API. At this level, the capabilities

of D-Bus approach those of a bus system, as the name suggests. The server process, *dbus-daemon*, runs in the background and listens for connection requests by applications that register for specific event types – plugging and unplugging of hardware, for example. When an event occurs, the D-Bus daemon sends a message via the bus, and the application in question responds accordingly.

System Global or Per Session

Basically, systems that use D-Bus have two buses implemented by a single server process: the system bus and the session bus. The system bus is launched at boot time and is active even if there are no users logged on to the system. A server process for the session bus is enabled after completing the GUI-based login for a desktop session. The *dbus-daemon* binary provides the *--system*

and *--session* command line parameters for these two modes.

To launch the daemon, the D-Bus package includes a *dbus-launch* tool, which sets up the required environmental variables, among other things. Most distributions launch the D-Bus daemon in session mode when launching an X session.

Figure 3 shows the role played by these two buses in communications between operating system components. The session buses gives applications belonging to a desktop session the ability to talk to each other. The applications can be services provided by the desktop environment. The system bus is mainly designed for helping desktop programs talk to the underlying layers. For example, an application can use the system bus to register with a hardware class such as digital cameras.

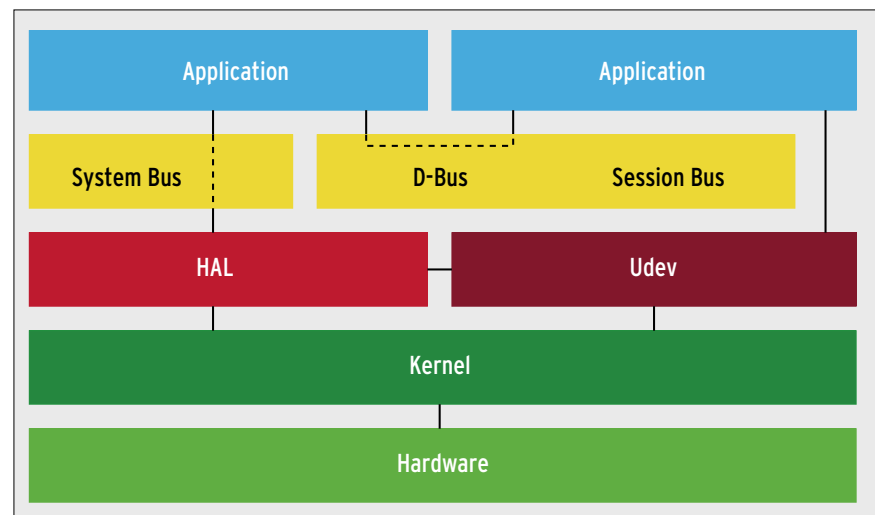


Figure 3: Component interactions follow this scheme.