Writing udev rules

by Daniel Drake (dsd)
Version 0.74

The most recent version of this document can always be found at:
http://www.reactivated.net/writing_udev_rules.html

Contents

Introduction

About this document

udev is targeted at Linux kernels 2.6 and beyond to provide a userspace solution for a dynamic /dev directory, with persistent device naming. The previous /dev implementation, *devfs*, is now deprecated, and udev is seen as the successor. udev vs devfs is a sensitive area of conversation - you should read this document before making comparisons.

Over the years, the things that you might use udev rules for has changed, as well as the flexibility of rules themselves. On a modern system, udev provides persistent naming for some device types out-of-the-box, eliminating the need for custom rules for those devices. However, some users will still require the extra level of customisation.

This document assumes that you have udev installed and running OK with default configurations. This is usually handled by your Linux distribution.

This document does not cover every single detail of rule writing, but does aim to introduce all of the main concepts. The finer details can be found in the udev man page.

This document uses various examples (many of which are entirely fictional) to illustrate ideas and concepts. Not all syntax is explicitly described in the accompanying text, be sure to look at the example rules to get a complete understanding.

History

- April 5th 2008 v0.74: Typo fixes.
- December 3rd 2007 v0.73: Update for new udev versions, and some miscellaneous improvements.
- October 2nd 2006 v0.72: Fixed a typo in one of the example rules.
- June 10th 2006 v0.71: Misc changes based on recent feedback - thanks!
- June 3rd 2006 v0.7: Complete rework, to be more suited for the modern-day udev.
- May 9th 2005 v0.6: Misc updates, including information about udevinfo, groups and permissions, logging, and udevtest.
- June 20th 2004 v0.55: Added info on multiple symlinks, and some minor changes/updates.
- April 26th 2004 v0.54: Added some Debian info. Minor corrections. Re-reverted information about what to call your rule file. Added info about naming network interfaces.
- April 15th 2004 v0.53: Minor corrections. Added info about NAME{all_partitions}. Added info about other udevinfo tricks.
- April 14th 2004 v0.52: Reverted to suggesting using "udev.rules" until the udev defaults allow for other files. Minor work.
- April 6th 2004 v0.51: I now write suggest users to use their own "local.rules" file rather than prepending "udev.rules".
- April 3rd 2004 v0.5: Minor cleanups and preparations for possible inclusion in the udev distribution.
- March 20th 2004 v0.4: General improvements, clarifications, and cleanups. Added more information about writing rules for usb-storage.
- February 23rd 2004 v0.3: Rewrote some parts to emphasise how sysfs naming works, and how it can be matched. Updated rule-writing parts to represent udev 018s new SYSFS{filename} naming scheme. Improved sectioning, and clarified many points. Added info about KDE.
- February 18th 2004 v0.2: Fixed a small omission in an example. Updated section on identifying mass-storage devices. Updated section on nvidia.
- February 15th 2004 v0.1: Initial publication.

The concepts

Terminology: devfs, sysfs, nodes, etc.

A basic introduction only, might not be totally accurate.

On typical Linux-based systems, the */dev* directory is used to store file-like device **nodes** which refer to certain devices in the system. Each node points to a part of the system (a device), which might or might not exist. Userspace applications can use these device nodes to interface with the systems hardware, for example, the X server will "listen to" /dev/input/mice so that it can relate the user's mouse movements to moving the visual mouse pointer.

The original */dev* directories were just populated with every device that might possibly appear in the system. /dev directories were typically very large because of this. **devfs** came along to provide a more manageable approach (noticeably, it only populated /dev with hardware that is plugged into the system), as well as some other functionality, but the system proved to have problems which could not be easily fixed.

udev is the "new" way of managing */dev* directories, designed to clear up some issues with previous */dev* implementations, and provide a robust path forward. In order to create and name */dev* device nodes corresponding to devices that are present in the system, udev relies on matching information provided by *sysfs* with *rules* provided by the user. This documentation aims to detail the process of rule-writing, one of the only udev-related tasks that must (optionally) be performed by the user.

sysfs is a new filesystem to the 2.6 kernels. It is managed by the kernel, and exports basic information about the devices currently plugged into your system. udev can use this information to create device nodes corresponding to your hardware. sysfs is mounted at */sys* and is browseable. You may wish to investigate some of the files stored there before getting to grips with udev. Throughout this document, I will use the terms */sys* and *sysfs* interchangeably.

Why?

udev rules are flexible and very powerful. Here are some of the things you can use rules to achieve:

- Rename a device node from the default name to something else
- Provide an alternative/persistent name for a device node by creating a symbolic link to the default device node
- Name a device node based on the output of a program
- Change permissions and ownership of a device node
- Launch a script when a device node is created or deleted (typically when a device is attached or unplugged)
- Rename network interfaces

Writing rules is <u>not</u> a workaround for the problem where no device nodes for your particular device exist. Even if there are no matching rules, udev will create the device node with the default name supplied by the kernel.

Having persistently named device nodes has several advantages. Assume you own two USB storage devices: a digital camera and a USB flash disk. These devices are typically assigned device nodes */dev/sda* and */dev/sdb* but the exact assignment depends on the order which they were originally connected. This may cause problems to some users, who would benefit greatly if each device could be named persistently every time, e.g. */dev/camera* and */dev/flashdisk*.

Built-in persistent naming schemes

udev provides persistent naming for some device types out of the box. This is a very useful feature, and in many circumstances means that your journey ends here: you do not have to write any rules.

udev provides out-of-the-box persistent naming for storage devices in the */dev/disk* directory. To view the persistent names which have been created for your storage hardware, you can use the following command:

```
# ls -lR /dev/disk
```

This works for all storage types. As an example, udev has created */dev/disk/by-id/scsi-SATA_ST3120827AS_4MS1NDXZ-part3* which is a persistent-named symbolic link to my root partition. udev creates */dev/disk/by-id/usb-Prolific_Technology_Inc._USB_Mass_Storage_Device-part1* when I plug my USB flash disk in, which is also a persistent name.

Rule writing

Rule files and semantics

When deciding how to name a device and which additional actions to perform, udev reads a series of rules files. These files are kept in the */etc/udev/rules.d* directory, and they all must have the *.rules* suffix.

Default udev rules are stored in */etc/udev/rules.d/50-udev.rules*. You may find it interesting to look over this file - it includes a few examples, and then some default rules proving a devfs-style /dev layout. However, you should not write rules into this file directly.

Files in /etc/udev/rules.d/ are parsed in **lexical** order, and in some circumstances, the order in which rules are parsed is important. In general, you want your own rules to be parsed before the defaults, so I suggest you create a file at */etc/udev/rules.d/10-local.rules* and write all your rules into this file.

In a rules file, lines starting with "#" are treated as comments. Every other non-blank line is a rule. Rules cannot span multiple lines.

One device can be matched by more than one rule. This has it's practical advantages, for example, we can write two rules which match the same device, where each one provides its own alternate name for the device. Both alternate names will be created, even if the rules are in separate files. It is important to understand that udev will *not* stop processing when it finds a matching rule, it will continue searching and attempt to apply every rule that it knows about.

Rule syntax

Each rule is constructed from a series of key-value pairs, which are separated by commas. **match** keys are conditions used to identify the device which the rule is acting upon. When all match keys in a rule correspond to the device being handled, then the rule is applied and the actions of the **assignment** keys are invoked. Every rule should consist of at least one match key and at least one assignment key.

Here is an example rule to illustrate the above:

```
KERNEL=="hdb", NAME="my_spare_disk"
```

The above rule includes one match key (*KERNEL*) and one assignment key (*NAME*). The semantics of these keys and their properties will be detailed later. It is important to note that the match key is related to its value through the equality operator (==), whereas the assignment key is related to its value through the assignment operator (=).

Be aware that udev does not support any form of line continuation. Do not insert any line breaks in your rules, as this will cause udev to see your one rule as multiple rules and will not work as expected.

Basic Rules

udev provides several different match keys which can be used to write rules which match devices very precisely. Some of the most common keys are introduced below, others will be introduced later in this document. For a complete list, see the udev man page.

- **KERNEL** - match against the kernel name for the device
- **SUBSYSTEM** - match against the subsystem of the device
- **DRIVER** - match against the name of the driver backing the device

After you have used a series of match keys to precisely match a device, udev gives you fine control over what happens next, through a range of assignment keys. For a complete list of possible assignment keys, see the udev man page. The most basic assignment keys are introduced below. Others will be introduced later in this document.

- **NAME** - the name that shall be used for the device node
- **SYMLINK** - a **list** of symbolic links which act as alternative names for the device node

As hinted above, udev only creates one true device node for one device. If you wish to provide alternate names for this device node, you use the symbolic link functionality. With the *SYMLINK* assignment, you are actually maintaining a *list* of symbolic links, all of which will be pointed at the real device node. To manipulate these links, we introduce a new operator for appending to lists: +=. You can append multiple symlinks to the list from any one rule by separating each one with a space.

```
KERNEL=="hdb", NAME="my_spare_disk"
```

The above rule says: *match a device which was named by the kernel as hdb, and instead of calling it hdb, name the device node as my_spare_disk*. The device node appears at */dev/my_spare_disk*.

```
KERNEL=="hdb", DRIVER=="ide-disk", SYMLINK+="sparedisk"
```

The above rule says: *match a device which was named by the kernel as hdb AND where the driver is ide-disk. Name the device node with the default name and create a symbolic link to it named sparedisk*. Note that we did not specify a device node name, so udev uses the default. In order to preserve the standard */dev* layout, your own rules will typically leave the NAME alone but create some SYMLINKs and/or perform other assignments.

```
KERNEL=="hdc", SYMLINK+="cdrom cdrom0"
```

The above rule is probably more typical of the types of rules you might be writing. It creates two symbolic links at */dev/cdrom* and */dev/cdrom0*, both of which point at */dev/hdc*. Again, no NAME assignment was specified, so the default kernel name (hdc) is used.

Matching sysfs attributes

The match keys introduced so far only provide limited matching capabilities. Realistically we require much finer control: we want to identify devices based on advanced properties such as vendor codes, exact product numbers, serial numbers, storage capacities, number of partitions, etc.

Many drivers export information like this into sysfs, and udev allows us to incorporate sysfs-matching into our rules, using the *ATTR* key with a slightly different syntax.

Here is an example rule which matches a single attribute from sysfs. Further detail will be provided later in this document which will aid you in writing rules based on sysfs attributes.

```
SUBSYSTEM=="block", ATTR{size}=="234441648", SYMLINK+="my_disk"
```

Device hierarchy

The Linux kernel actually represents devices in a tree-like structure, and this information is exposed through sysfs and useful when writing rules. For example, the device representation of my hard disk device is a child of the SCSI disk device, which is in turn a child of the Serial ATA controller device, which is in turn a child of the PCI bus device. It is likely that you will find yourself needing to refer to information from a parent of the device in question, for example the serial number of my hard disk device is not exposed at the device level, it is exposed by its direct parent at the SCSI disk level.

The four main match keys introduced so far (KERNEL/SUBSYSTEM/DRIVER/ATTR) only match against values corresponding to the device in question, and do not match values from parent devices. udev provides variants of the match keys that will search upwards through the tree:

- **KERNELS** - match against the kernel name for the device, or the kernel name for any of the parent devices
- **SUBSYSTEMS** - match against the subsystem of the device, or the subsystem of any of the parent devices
- **DRIVERS** - match against the name of the driver backing the device, or the name of the driver backing any of the parent devices
- **ATTRS** - match a sysfs attribute of the device, or a sysfs attribute of any of the parent devices

With hierarchy considerations in mind, you may feel that rule writing is becoming a little complicated. Rest assured that there are tools that help out here, which will be introduced later.

String substitutions

When writing rules which will potentially handle multiple similar devices, udev's *printf-like string substitution operators* are very useful. You can simply include these operators in any assignments your rule makes, and udev will evaluate them when they are executed.

The most common operators are **%k** and **%n**. %k evaluates to the kernel name for the device, e.g. "sda3" for a device that would (by default) appear at */dev/sda3*. **%n** evaluates to the kernel number for the device (the partition number for storage devices), e.g. "3" for */dev/sda3*.

udev also provides several other substitution operators for more advanced functionality. Consult the udev man page after reading the rest of this document. There is also an alternative syntax for these operators - **$kernel** and **$number** for the examples above. For this reason, if you wish to match a literal % in a rule then you must write **%%**, and if you wish to match a literal $ then you must write **$$**.

To illustrate the concept of string substitution, some example rules are shown below.

```
KERNEL=="mice", NAME="input/%k"
KERNEL=="loop0", NAME="loop/%n", SYMLINK+="%k"
```

The first rule ensures that the mice device node appears exclusively in the */dev/input* directory (by default it would be at */dev/mice*). The second rule ensures that the device node named loop0 is created at */dev/loop/0* but also creates a symbolic link at */dev/loop0* as usual.

The use of the above rules is questionable, as they all could be rewritten without using any substitution operators. The true power of these substitutions will become apparent in the next section.

String matching

As well as matching strings exactly, udev allows you to use shell-style pattern matching. There are 3 patterns supported:

- ***** - match any character, zero or more times
- **?** - match any character exactly once
- **[]** - match any single character specified in the brackets, ranges are also permitted

Here are some examples which incorporate the above patterns. Note the use of the string substitution operators.

```
KERNEL=="fd[0-9]*", NAME="floppy/%n", SYMLINK+="%k"
KERNEL=="hiddev*", NAME="usb/%k"
```

The first rule matches all floppy disk drives, and ensures that the device nodes are placed in the */dev/floppy* directory, as well as creating a symbolic link from the default name. The second rule ensures that hiddev devices are only present in the */dev/usb* directory.

Finding information from sysfs

The sysfs tree

The concept of using interesting information from sysfs was briefly touched upon above. In order to write rules based on this information, you first need to know the names of the attributes and their current values.

sysfs is actually a very simple structure. It is logically divided into directories. Each directory contains a number of files (*attributes*) which typically contain just one value. Some symbolic links are present, which link devices to their parents. The hierarchical structure was touched upon above.

Some directories are referred to as *top-level device paths*. These directories represent actual devices that have corresponding device nodes. Top-level device paths can be classified as sysfs directories which contain a *dev* file, the following command will list these for you:

```
# find /sys -name dev
```

For example, on my system, the */sys/block/sda* directory is the device path for my hard disk. It is linked to it's parent, the SCSI disk device, through the */sys/block/sda/device* symbolic link.

When you write rules based on sysfs information, you are simply matching attribute contents of some files in one part of the chain. For example, I can read the size of my hard disk as follows:

```
# cat /sys/block/sda/size
234441648
```

In a udev rule, I could use ATTR{size}=="234441648" to identify this disk. As udev iterates through the entire device chain, I could alternatively opt to match attributes in another part of the chain (e.g. attributes in */sys/class/block/sda/device/*) using *ATTRS*, however there are some caveats when dealing with different parts of the chain which are described later.

Although this serves as a useful introduction as to the structure of sysfs and exactly how udev matches values, manually trawling through sysfs is both time consuming and unnecessary.

udevinfo

Enter *udevinfo*, which is probably the most straightforward tool you can use to construct rules. All you need to know is the sysfs device path of the device in question. A trimmed example is shown below:

```
# udevinfo -a -p /sys/block/sda

  looking at device '/block/sda':
    KERNEL=="sda"
    SUBSYSTEM=="block"
    ATTR{stat}=="  128535     2246 2788977  766188     73998   317300 3132216 5735004       0  516516 6503316"
    ATTR{size}=="234441648"
    ATTR{removable}=="0"
    ATTR{range}=="16"
    ATTR{dev}=="8:0"

  looking at parent device '/devices/pci0000:00/0000:00:07.0/host0/target0:0:0/0:0:0:0':
    KERNELS=="0:0:0:0"
    SUBSYSTEMS=="scsi"
    DRIVERS=="sd"
    ATTRS{ioerr_cnt}=="0x0"
    ATTRS{iodone_cnt}=="0x31737"
    ATTRS{iorequest_cnt}=="0x31737"
    ATTRS{iocounterbits}=="32"
    ATTRS{timeout}=="30"
    ATTRS{state}=="running"
    ATTRS{rev}=="3.42"
    ATTRS{model}=="ST3120827AS     "
    ATTRS{vendor}=="ATA     "
    ATTRS{scsi_level}=="6"
    ATTRS{type}=="0"
    ATTRS{queue_type}=="none"
    ATTRS{queue_depth}=="1"
    ATTRS{device_blocked}=="0"

  looking at parent device '/devices/pci0000:00/0000:00:07.0':
    KERNELS=="0000:00:07.0"
    SUBSYSTEMS=="pci"
    DRIVERS=="sata_nv"
    ATTRS{vendor}=="0x10de"
    ATTRS{device}=="0x037f"
```

As you can see, udevinfo simply produces a list of attributes you can use as-is as match keys in your udev rules. From the above example, I could produce (e.g.) either of the following two rules for this device:

```
SUBSYSTEM=="block", ATTR{size}=="234441648", NAME="my_hard_disk"
SUBSYSTEM=="block", SUBSYSTEMS=="scsi", ATTRS{model}=="ST3120827AS", NAME="my_hard_disk"
```

You may have noted the use of colour in the above examples. This is to demonstrate that while it is legal to combine the attributes from the device in question and a *single* parent device, you cannot mix-and-match attributes from multiple parent devices - your rule will not work. For example, the following rule is *invalid* as it attempts to match attributes from two parent devices:

```
SUBSYSTEM=="block", ATTRS{model}=="ST3120827AS", DRIVERS=="sata_nv", NAME="my_hard_disk"
```

You are usually provided with a large number of attributes, and you must pick a number of them to construct your rule. In general, you want to choose attributes which identify your device in a persistent and human-recognisable way. In the examples above, I chose the size of my disk and its model number. I did not use meaningless numbers such as ATTRS{iodone_cnt}=="0x31737".

Observe the effects of hierarchy in the udevinfo output. The green section corresponding to the device in question uses the standard match keys such as KERNEL and ATTR. The blue and maroon sections corresponding to parent devices use the parent-traversing variants such as SUBSYSTEMS and ATTRS. This is why the complexity introduced by the hierarchical structure is actually quite easy to deal with, just be sure to use the exact values that udevinfo suggests.

Another point to note is that it is common for text attributes to appear in the udevinfo output to be padded with spaces (e.g. see ST3120827AS above). In your rules, you can either specify the extra spaces, or you can cut them off as I have done.

The only complication with using udevinfo is that you are required to know the top-level device path (/sys/block/sda in the example above). This is not always obvious. However, as you are generally writing rules for device nodes which already exist, you can use udevinfo to look up the device path for you:

```
# udevinfo -a -p $(udevinfo -q path -n /dev/sda)
```

Alternative methods

Although udevinfo is almost certainly the most straightforward way of listing the exact attributes you can build rules from, some users are happier with other tools. Utilities such as usbview display a similar set of information, most of which can be used in rules.

Advanced topics

Controlling permissions and ownership

udev allows you to use additional assignments in rules to control ownership and permission attributes on each device.

The *GROUP* assignment allows you to define which Unix group should own the device node. Here is an example rule which defines that the *video* group will own the framebuffer devices:

```
KERNEL=="fb[0-9]*", NAME="fb/%n", SYMLINK+="%k", GROUP="video"
```

The *OWNER* key, perhaps less useful, allows you to define which Unix user should have ownership permissions on the device node. Assuming the slightly odd situation where you would want *john* to own your floppy devices, you could use:

```
KERNEL=="fd[0-9]*", OWNER="john"
```

udev defaults to creating nodes with Unix permissions of 0660 (read/write to owner and group). If you need to, you can override these defaults on certain devices using rules including the *MODE* assignment. As an example, the following rule defines that the inotify node shall be readable and writable to everyone:

```
KERNEL=="inotify", NAME="misc/%k", SYMLINK+="%k", MODE="0666"
```

Using external programs to name devices

Under some circumstances, you may require more flexibility than standard udev rules can provide. In this case, you can ask udev to run a program and use the standard output from that program to provide device naming.

To use this functionality, you simply specify the absolute path of the program to run (and any parameters) in the *PROGRAM* assignment, and you then use some variant of the *%c* substitution in the NAME/SYMLINK assignments.

The following examples refer to a fictional program found at */bin/device_namer*. device_namer takes one command line argument which is the kernel name for the device. Based upon this kernel name, device_namer does its magic and produces some output to the usual *stdout* pipe, split into several parts. Each part is just a single word, and parts are separated by a single space.

In our first example, we assume that device_namer outputs a number of parts, each one to form a symbolic link (alternative name) for the device in question.

```
KERNEL=="hda", PROGRAM="/bin/device_namer %k", SYMLINK+="%c"
```

The next example assumes that device_namer outputs two parts, the first being the device name, and the second being the name for an additional symbolic link. We now introduce the *%c{N}* substitution, which refers to part N of the output:

```
KERNEL=="hda", PROGRAM="/bin/device_namer %k", NAME="%c{1}", SYMLINK+="%c{2}"
```

The next example assumes that device_namer outputs one part for the device name, followed by any number of parts which will form additional symbolic links. We now introduce the *%c{N+}* substitution, which evaluates to part N, N+1, N+2, ... until the end of the output.

```
KERNEL=="hda", PROGRAM="/bin/device_namer %k", NAME="%c{1}", SYMLINK+="%c{2+}"
```

Output parts can be used in any assignment key, not only NAME and SYMLINK. The example below uses a fictional program to determine the Unix group which should own the device:

```
KERNEL=="hda", PROGRAM="/bin/who_owns_device %k", GROUP="%c"
```

Running external programs upon certain events

Yet another reason for writing udev rules is to run a particular program when a device is connected or disconnected. For example, you might want to execute a script to automatically download all of your photos from your digital camera when it is connected.

Do not confuse this with the *PROGRAM* functionality described above. *PROGRAM* is used for running programs which produce device names (and they shouldn't do anything other than that). When those programs are being executed, the device node has not yet been created, so acting upon the device in any way is not possible.

The functionality introduced here allows you to run a program after the device node is put in place. This program can act on the device, however it must not run for any extended period of time, because udev is effectively paused while these programs are running. One workaround for this limitation is to make sure your program immediately detaches itself.

Here is an example rule which demonstrates the use of the *RUN* list assignment:

```
KERNEL=="sdb", RUN+="/usr/bin/my_program"
```

When */usr/bin/my_program* is executed, various parts of the udev environment are available as environment variables, including key values such as *SUBSYSTEM*. You can also use the *ACTION* environment variable to detect whether the device is being connected or disconnected - ACTION will be either "add" or "remove" respectively.

udev does not run these programs on any active terminal, and it does not execute them under the context of a shell. Be sure to ensure your program is marked executable, if it is a shell script ensure it starts with an appropriate [shebang](#) (e.g. #!/bin/sh), and do not expect any standard output to appear on your terminal.

Environment interaction

udev provides an *ENV* key for environment variables which can be used for both matching and assignment.

In the assignment case, you can set environment variables which you can then match against later. You can also set environment variables which can be used by any external programs invoked using the techniques mentioned above. A fictional example rule which sets an environment variable is shown below.

```
KERNEL="fd0", SYMLINK+="floppy", ENV{some_var}="value"
```

In the matching case, you can ensure that rules only run depending on the value of an environment variable. Note that the environment that udev sees will not be the same user environment as you get on the console. A fictional rule involving an environment match is shown below.

```
KERNEL="fd0", ENV{an_env_var}=="yes", SYMLINK+="floppy"
```

The above rule only creates the */dev/floppy* link if $an_env_var is set to "yes" in udev's environment.

Additional options

Another assignment which can prove useful is the *OPTIONS* list. A few options are available:

- **all_partitions** - create all possible partitions for a block device, rather than only those that were initially detected
- **ignore_device** - ignore the event completely
- **last_rule** - ensure that no later rules have any effect

For example, the rule below sets the group ownership on my hard disk node, and ensures that no later rule can have any effect:

```
KERNEL=="sda", GROUP="disk", OPTIONS+="last_rule"
```

Examples

USB Printer

I power on my printer, and it is assigned device node */dev/lp0*. Not satisfied with such a bland name, I decide to use udevinfo to aid me in writing a rule which will provide an alternative name:

```
# udevinfo -a -p $(udevinfo -q path -n /dev/lp0)
  looking at device '/class/usb/lp0':
    KERNEL=="lp0"
    SUBSYSTEM=="usb"
    DRIVER==""
    ATTR{dev}=="180:0"

  looking at parent device '/devices/pci0000:00/0000:00:1d.0/usb1/1-1':
    SUBSYSTEMS=="usb"
    ATTRS{manufacturer}=="EPSON"
    ATTRS{product}=="USB Printer"
    ATTRS{serial}=="L72010011070626380"
```

My rule becomes:

```
SUBSYSTEM=="usb", ATTRS{serial}=="L72010011070626380", SYMLINK+="epson_680"
```

USB Camera

Like most, my camera identifies itself as an external hard disk connected over the USB bus, using the SCSI transport. To access my photos, I mount the drive and copy the image files onto my hard disk.

Not all cameras work in this way: some of them use a non-storage protocol such as cameras supported by gphoto2. In the gphoto case, you do not want to be writing rules for your device, as is it controlled purely through userspace (rather than a specific kernel driver).

A common complication with USB camera devices is that they usually identify themselves as a disk with a single partition, in this case */dev/sdb* with */dev/sdb1*. The sdb node is useless to me, but sdb1 is interesting - this is the one I want to mount. There is a problem here that because sysfs is chained, the useful attributes which udevinfo produces for /dev/sdb1 are identical to the ones for /dev/sdb. This results in your rule potentially matching <u>both</u> the raw disk and the partition, which is not what you want, your rule should be **specific**.

To get around this, you simply need to think about what differs between sdb and sdb1. It is surprisingly simple: the name itself differs, so we can use a simple pattern match on the NAME field.

```
# udevinfo -a -p $(udevinfo -q path -n /dev/sdb1)
  looking at device '/block/sdb/sdb1':
    KERNEL=="sdb1"
    SUBSYSTEM=="block"

  looking at parent device '/devices/pci0000:00/0000:00:02.1/usb1/1-1/1-1:1.0/host6/target6:0:0/6:0:0:0':
    KERNELS=="6:0:0:0"
    SUBSYSTEMS=="scsi"
    DRIVERS=="sd"
    ATTRS{rev}=="1.00"
    ATTRS{model}=="X250,D560Z,C350Z"
    ATTRS{vendor}=="OLYMPUS "
    ATTRS{scsi_level}=="3"
    ATTRS{type}=="0"
```

My rule:

```
KERNEL=="sd?1", SUBSYSTEMS=="scsi", ATTRS{model}=="X250,D560Z,C350Z", SYMLINK+="camera"
```

USB Hard Disk

A USB hard disk is comparable to the USB camera I described above, however typical usage patterns are different. In the camera example, I explained that I am not interested in the sdb node - it's only real use is for partitioning (e.g. with fdisk), but why would I want to partition my camera!?

Of course, if you have a 100GB USB hard disk, it is perfectly understandable that you might want to partition it, in which case we can take advantage of udev's string substitutions:

```
KERNEL=="sd*", SUBSYSTEMS=="scsi", ATTRS{model}=="USB 2.0 Storage Device", SYMLINK+="usbhd%n"
```

This rule creates symlinks such as:

- */dev/usbhd* - The fdiskable node
- */dev/usbhd1* - The first partition (mountable)
- */dev/usbhd2* - The second partition (mountable)

USB Card Reader

USB card readers (CompactFlash, SmartMedia, etc) are yet another range of USB storage devices which have different usage requirements.

These devices typically do not inform the host computer upon media change. So, if you plug in the device with no media, and then insert a card, the computer does not realise, and you do not have your mountable sdb1 partition node for the media.

One possible solution is to take advantage of the *all_partitions* option, which will create 16 partition nodes for every block device that the rule matches:

```
KERNEL="sd*", SUBSYSTEMS=="scsi", ATTRS{model}=="USB 2.0 CompactFlash Reader", SYMLINK+="cfrdr%n", OPTIONS+="all_partitions"
```

You will now have nodes named: cfrdr, cfrdr1, cfrdr2, cfrdr3, ..., cfrdr15.

USB Palm Pilot

These devices work as USB-serial devices, so by default, you only get the *ttyUSB1* device node. The palm utilities rely on */dev/pilot*, so many users will want to use a rule to provide this.

Carsten Clasohm's blog post appears to be the definitive source for this. Carsten's rule is shown below:

```
SUBSYSTEMS=="usb", ATTRS{product}=="Palm Handheld", KERNEL=="ttyUSB*", SYMLINK+="pilot"
```

Note that the product string seems to vary from product to product, so make sure that you check (using udevinfo) which one applies to you.

CD/DVD drives

I have two optical drives in this computer: a DVD reader (hdc), and a DVD rewriter (hdd). I do not expect these device nodes to change, unless I physically rewire my system. However, many users like to have device nodes such as */dev/dvd* for convenience.

As we know the KERNEL names for these devices, rule writing is simple. Here are some examples for my system:

```
SUBSYSTEM=="block", KERNEL=="hdc", SYMLINK+="dvd", GROUP="cdrom"
SUBSYSTEM=="block", KERNEL=="hdd", SYMLINK+="dvdrw", GROUP="cdrom"
```

Network interfaces

Even though they are referenced by names, network interfaces typically do not have device nodes associated with them. Despite that, the rule writing process is almost identical.

It makes sense to simply match the MAC address of your interface in the rule, as this is unique. However, make sure that you use the *exact* MAC address as shown as udevinfo, because if you do not match the case exactly, your rule will not work.

```
# udevinfo -a -p /sys/class/net/eth0
  looking at class device '/sys/class/net/eth0':
    KERNEL=="eth0"
    ATTR{address}=="00:52:8b:d5:04:48"
```

Here is my rule:

```
KERNEL=="eth*", ATTR{address}=="00:52:8b:d5:04:48", NAME="lan"
```

You will need to reload the net driver for this rule to take effect. You can either unload and reload the module, or simply reboot the system. You will also need to reconfigure your system to use "lan" rather than "eth0". I had some troubles getting this going (the interface wasn't being renamed) until I had completely dropped all references to eth0. After that, you should be able to use "lan" instead of "eth0" in any calls to ifconfig or similar utilities.

Testing and debugging

Putting your rules into action

Assuming you are on a recent kernel with *inotify* support, udev will automatically monitor your rules directory and automatically pick up any modifications you make to the rule files.

Despite this, udev will not automatically reprocess all devices and attempt to apply the new rule(s). For example, if you write a rule to add an extra symbolic link for your camera while your camera is plugged in, you cannot expect the extra symbolic link to show up right away.

To make the symbolic link show up, you can either disconnect and reconnect your camera, or alternatively in the case of non-removable devices, you can run **udevtrigger**.

If your kernel does not have inotify support, new rules will not be detected automatically. In this situation, you must run **udevcontrol reload_rules** after making any rule file modifications for those modifications to take effect.

udevtest

If you know the top-level device path in sysfs, you can use **udevtest** to show the actions which udev would take. This may help you debug your rules. For example, assuming you want to debug a rule which acts on */sys/class/sound/dsp*:

```
# udevtest /class/sound/dsp
main: looking at device '/class/sound/dsp' from subsystem 'sound'
udev_rules_get_name: add symlink 'dsp'
udev_rules_get_name: rule applied, 'dsp' becomes 'sound/dsp'
udev_device_event: device '/class/sound/dsp' already known, remove possible symlinks
udev_node_add: creating device node '/dev/sound/dsp', major = '14', minor = '3', mode = '0660', uid = '0', gid = '18'
udev_node_add: creating symlink '/dev/dsp' to 'sound/dsp'
```

Note the */sys* prefix was removed from the udevtest command line argument, this is because udevtest operates on device paths. Also note that udevtest is purely a testing/debugging tool, it does not create any device nodes, despite what the output suggests!

Author and contact

This document is written by Daniel Drake <dan@reactivated.net>. Feedback is appreciated.

For support, you should mail the linux-hotplug mailing list: linux-hotplug-devel@lists.sourceforge.net.