Extending hotplug on Debian, SLES 9, and RHAS 4

JUMP START

Debian hotplug is designed for little more than loading drivers and configuring devices. The collection of scripts discussed in this article helps Linux to respond when a network cable is plugged in or when a cellphone is in the vicinity. Read on for more on how to define custom hotplug events. **BY MIRKO DÖLLE**

On Debian Sage, Suse Enterprise Linux 9, Red Hat Advanced Server 4, and older systems, the hotplug daemon is the central contact point for all device events. However, the daemon does little more than load drivers for new devices – even the simple task of responding when a network cable is plugged or unplugged is too much for the implementation used by these distributions.

It is hard to understand why the built-in hotplug configuration is so limited; after all, kernel hotplugging is easily capable of performing a fully automated backup on a USB stick plugged in by a user, or updating the MP3 collection on a cellphone in the vicinity, using a few simple scripts. This article describes some tricks and scripts you can use to extend the hotplug system.

Cable Control

netplugd [1] handles link detection for network connections. The developer, Bryan O'Sullivan, designed the daemon to be called by *init*. After launching, a *netplugd* process monitors each connector. A bash script called */etc/netplug.d/netplug* defines how *netplugd* should respond to a link state change for a connection.

But hotplug itself, or rather the */etc/hotplug/net.agent* script, is actually responsible for configuring the network device. In the case of Ethernet devices, the script launches *ifup*, which either assigns a static IP address or uses DHCP to request an address. Additionally, the *init* script, */etc/init.d/networking*, calls *ifup* when the system launches, and sets up all the network devices listed in */etc/network/interfaces*.

This approach does not make much sense and causes considerable delays. If a user boots a laptop without a network connection, for example, the DHCP client waits for a response that it can't pos-

Bus Drivers

Assuming a kernel with *CONFIG_HOTPLUG* built in, the */proc/sys/kernel/hotplug* pseudo file specifies which daemon the kernel should call when a specific bus driver registers a new device with the system (Figure 1). This does not necessarily mean that the device has just been physically attached to the system; instead, hotplug is just told that a new device has become available.

For example, the tg3 network card driver can trigger a hotplug event when it is loaded, assuming it finds the matching Broadcom hardware. Thus, hotplug is not only responsible for hotplug events, but also for cold pluggin, that is, loading or unloading of the corresponding drivers, although no physical change to the computer has occurred.

Even if a device has hotplugging capabilities, kernel hotplugging does not rely on these capabilities. The best example of this is a network card: according to the specifications, it is entirely possible to insert or remove a network cable on the fly, or to connect the computer to a completely different network. Although most drivers are capable of detecting a link to a switch or some other network device, they do not pass this information on to hotplug. The reason for this is that NIC drivers are not bus system drivers, and only the bus drivers register hotplug events.

sibly receive. Additionally, an administrator will not typically want to distribute the configuration over too many locations. But the hotplug daemon helps to solve both of these problems.

Hotplug Class

First of all, you need to tone down the call to *ifup* in the *init* script. As every network driver will trigger a hotplug event as soon as it is loaded and the hardware has initialized, we only want *ifup -a* in the *init* script to configure the loopback devices – these are the only network devices not to trigger a hotplug event. To allow this to happen, Listing 1 modifes the */etc/network/interfaces* file to assign Ethernet devices to the *hotplug* class, to prevent automatic configuration. The listing shows the entries for a machine with two NICs, one of which uses DHCP, and the other of which has a static address assignment.

The *allow-hotplug* keyword, instead of the standard *auto*, tells *ifup -a* to ignore these network devices. When the hotplug daemon is called, the network drivers pass in the *ACTION* and *INTERFACE* environmental variables. *ACTION* has a value of either *register* or *unregister*, depending on whether the driver is adding or removing the network device, whereas *INTERFACE* contains the value of the network device in question, such as *eth0* or *eth1*.

If you have a static network configuration, like the configuration for *eth1* in Listing 1, it is debatable whether hotplug should even configure the device when it becomes available, or whether the device should wait until a network cable is attached. For servers, it can be quite useful to assign an IP address and routing information to an interface, if the server provides services such as Bind and CUPS, in order to bind these services to a static IP address when booting the system.

New Network Agent

It makes more sense for the system to wait for a network connection before attempting to configure network adapters that request addresses from external entities. Listing 2 shows an excerpt from a modified network agent; the agent handles dynamically configured hotplug network interfaces.

In Lines 6 through 10, the agent determines whether to configure the *INTERFACE* automatically, or whether this is a hotplug interface. This is done for reasons of compatibility, and it gives administrators the ability to set up Ethernet devices independently of hotplug.

Lines 11 through 13 determine whether the device has a static configuration; note the *static* keyword in the corresponding line in */etc/network/interfaces* – in this case, we want hotplug to configure the interface without inspecting the link status.

We want *netplugd* to monitor the status of network interfaces that have dynamic address assignments. Lines 14 through 22 handle this; they also ensure that the program is loaded. Line 17 is a special case: it assigns an IP address of 0.0.0.0 to the interface and enables the interface – this is done to remove any prior address assignments, and to ensure that *netplugd* will detect any changes to the link status.

Although the link LED might still be lit on an inactive network interface (Figure 2), the driver will note that the interface is inactive, and it will not report the link status or detect the transmission speed. The command in Line 17 allows the status to be passed in, without the interface having a valid IP address.

Double Gateway

Lines 23 through 29 set up the hotplug interfaces with static IP addresses, as specified in */etc/network/interfaces*,

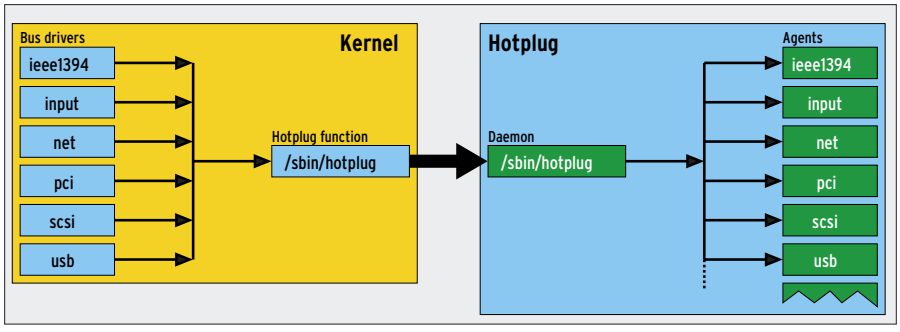

Figure 1: When bus drivers register a new device on the system, the kernel uses its hotplug function to launch the hotplug daemon. Hotplug then relies on the agent responsible for the bus system.

Listing 1: Interface Classes

```
01 auto lo
02 iface lo inet loopback
03
04 allow-hotplug eth0
05 iface eth0 inet dhcp
06
07 allow-hotplug eth1
08 iface eth1 inet static
09   address 192.168.178.110
10   netmask 255.255.255.0
11   network 192.168.178.0
12   broadcast 192.168.178.255
13   gateway 192.168.178.1
```

however, this approach can become an issue if the static configuration specifies a standard gateway (as in Listing 1), whereas another network device takes its network configuration from the DHCP server.

In this example, the standard gateway is reachable via *eth1* even if there is no network cable attached to the interface. If a DHCP server uses the network cable attached to *eth0* to assign both the IP address and a new default gateway, this will appear in the routing table, leading to a situation with two default gateways, and thus to massive routing problems.

If you have a laptop or some other portable, it doesn't make sense to configure the network interface unless a cable is attached, even if you have static address assignments. The changes necessary for handling this conditional configuration are quite minimal; just drop the code block between Lines 23 and 29, remove the test condition in Lines 11 through 13, and change Line 14 to the following:

Listing 2: Network Agent

```
01 #!/bin/sh
02 (...)
03 case $ACTION in
04 add|register)
05 (...)
06 if grep -q "^auto[[:
   space:]].*${INTERFACE}" /etc/
   network/interfaces; then
07   autoif=true
08 elif grep -q
   "^allow-hotplug[[:
   space:]].*${INTERFACE}" /etc/
   network/interfaces; then
09   hotplugif=true
10 fi
11 if grep -q "^iface[[:
   space:]]*${INTERFACE}[[:
   space:]]*inet[[:
   space:]]*static" /etc/network/
   interfaces; then
12   staticif=true
13 fi
14 if [ "${hotplugif}" = "true"
   -a "${staticif}" != "true" ];
   then
15   if [ -x /sbin/netplugd ];
   then
16     debug_mesg "iface
   $INTERFACE will be configured
   when link is active"
17     ifconfig ${INTERFACE}
   0.0.0.0 up
18     start-stop-daemon --start
   --background --pidfile /var/
   run/netplugd.${INTERFACE} \
19     --exec /sbin/netplugd --
   -i ${INTERFACE} -p /var/run/
   netplugd.${INTERFACE}
20   else
21     mesg "E: /sbin/netplugd
   not found. You need to install
   netplugd."
22   fi
23   exit 0
24 elif [ "${hotplugif}" = "true"
   -a "${staticif}" = "true" ];
   then
25   if ps -C ifup ho args | grep
   -q "$INTERFACE"; then
26     debug_mesg "Already
   ifup-ing that interface"
27   else
28     start-stop-daemon --start
   --background --pidfile /var/
   run/hotplug.net.ifup.bogus \
29     --startas /etc/hotplug/
   net.ifup -- "$INTERFACE"
30   fi
31   exit 0
32 elif [ "${NET_AGENT_POLICY}" =
   "auto" -a "${autoif}" = "true"
   ]; then
33   IFUPARG="$INTERFACE"
34   if [ "$autoif" = "true" ];
   then
35     IFUPARG='\
   ('$INTERFACE'\|-a\|--all\)'
36   fi
37   if ps -C ifup ho args | grep
   -q "$IFUPARG"; then
38     debug_mesg "Already
   ifup-ing that interface"
39   else
40     start-stop-daemon --start
   --background --pidfile /var/
   run/hotplug.net.ifup.bogus \
41     --startas /etc/hotplug/
   net.ifup --
   "$INTERFACE$LIFACE"
42   fi
43   exit 0
44 fi
45 (...)
```

```
elif [ "${hotplugif}" ➥
= "true" ]; then
```

Lines 30 through 41 in Listing 2 are virtually unchanged in comparison to the network agent for Debian Sarge; they handle network devices with automated configuration. There is just one major difference to the code block in Lines 23 through 29.

The test to check whether *ifup* is running not only needs to investigate the current interface, as in Line 24, but has to take *ifup -a* and *ifup --all* into consideration – Line 35 takes care of this, after Lines 31 through 34 have set up a *grep* search pattern.

Modifying Netplug

The Netplug daemon is actually designed to work independently of hotplug. For each link event, the daemon calls the */etc/netplug.d/netplug* script and passes in the interface in question, along with the event, either *in* *out*.

Listing 3 shows the new Netplug script, which now passes link events in to hotplug using *net_link* as the bus name. The advantage of the approach used in this script is that it gives administrators the ability to configure responses to hotplug events centrally in hotplug.

It makes no difference to hotplug whether the kernel or Netplug triggers the events. As long as Netplug is running as *root*, Hotplug can always load any missing drivers, create devices, and configure network devices. As */sbin/hotplug* will only search for an agent in the */etc/hotplug* directory named after the current bus, it is no problem to add more pseudo bus systems (Figure 3).

Listing 4 shows how the new */etc/hotplug/net_link.agent* net link agent works. When the network cable is plugged, the agent checks (Line 6) to make sure that the device really is a hotplug device. Line 8 ensures that no other *ifup* process is attempting to configure the network interface. Line 11 then sets up the network interface using the *net.ifup* on Debian Sarge.

When the network cable is unplugged, the code block in Lines 16 through 26 is run. First, in Lines 19 through 21, the net link agent has to terminate the *netplugd* process that monitors the interface, and that also triggered the event. The agent

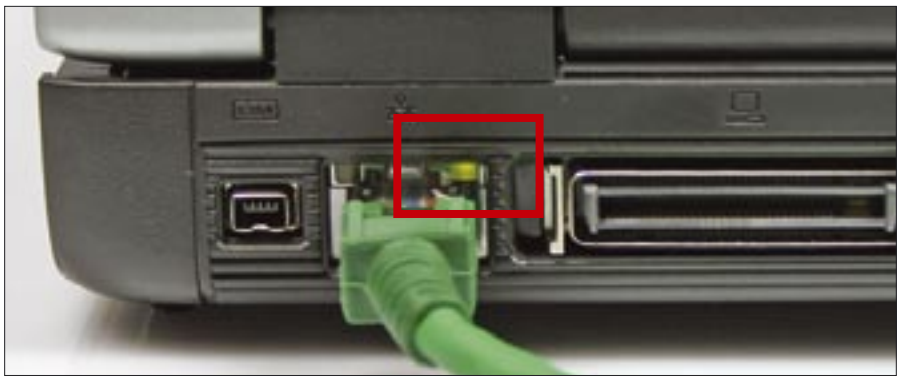

Figure 2: If the network device has been disabled via ifconfig or ifdown, the link LED may be lit when a cable is attached, however, the driver will not attempt to detect the network speed, or report that a cable has been plugged in.

then shuts down the network interface in Line 22 by calling *ifdown*.

Address Problems

As mentioned previously, a disabled network interface will not report any link events, and this makes it impossible for *netplugd* to react when a network cable is plugged. On the other hand, the Ethernet device will keep its old IP address until a new address is assigned. This makes it impossible to re-enable the interface simply by calling *ifconfig eth0 up*, as this would create a new entry for the device in the routing table.

To work around this problem, the net link agent configures the interface in Line 23 of Listing 4, assigning an IP address of *0.0.0.0* and enabling the interface. Of course, as *0.0.0.0* is an invalid IP address, this configuration has no effect on routing.

Listing 3: Netplug Script

```
01 #!/bin/bash
02 export INTERFACE="$1"
03 case "$2" in
04   in)
05     export ACTION="register"
06     /sbin/hotplug net_link
07     ;;
08   out)
09     export ACTION="unregister"
10     /sbin/hotplug net_link
11     ;;
12 esac
```

netplugd responds to the 0.0.0.0 address assignment by issuing the error message *unexpected state DONWANDOUT for UP* and terminating. To counteract this, the net link agent shuts down the service in Lines 19 through 21. To continue monitoring the network interface, and to reconfigure the interface as soon as a link is available, the *netplugd*

net link agent is then relaunched in Line 24.

In principle, *netplugd* supports wildcards such as *eth** when specifying the network interface. But as Netplug dies when the interface is disabled, it is important to run a Netplug daemon for each interface. Thus, you need a completely empty */etc/netplug/netplugd.conf* to pass the interface to be monitored to *netplugd* at the command line.

Bluetooth Events

Daemons such as *netplugd* that create events based on logical state changes are useful for many other tasks. For example you could use a daemon to detect a Bluetooth cellphone in its vicinity.

Listing 5 shows the *bluenear* daemon. It uses *hcitool* to attempt to set up a connection to a cellphone every five minutes. Once a link is established, it checks the link quality, which can be between 0 and 255. If the quality drops below 128, it assumes that the phone, and thus the user, are no longer near the computer, and this in turn trigers a hotplug event. A user agent can pick up the event and perform some action, such as locking the screen, for example.

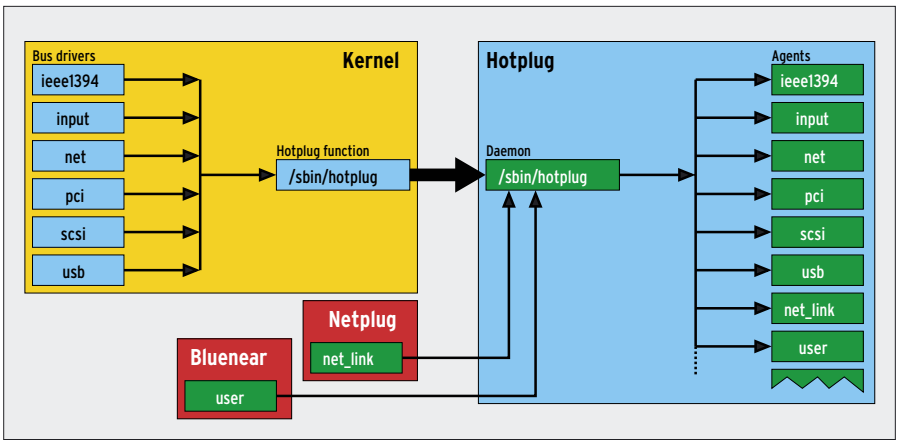

Figure 3: As long as the additional daemons are running as root, it makes no difference to hotplug whether the kernel or one of the daemons triggers the event. Additional agents are required for the new pseudo buses.

Lines 6 and 7 check if the Bluetooth device is within range, using a direct connection attempt, *hotplug cc*, followed by a call to the connection overview, *hotplug con*. The Bluetooth ID must be known and entered in Line 2 for this to happen. Although *hcitool scan* would list any Bluetooth devices in the vicinity, the *scan* parameter restricts the list to visible Bluetooth devices only.

A direct Bluetooth connection adds the advantage of being able to determine the signal strength – *hcitool* in Line 8. If this drops below the preset value, we can assume that the distance between the Bluetooth device and the computer is too great, and this results in the user being classified as away in Lines 13 through 15. The same thing happens in Lines 26 through 28 if it is impossible to establish a connection. As soon as the phone is in the vicinity, Lines 19 through 21 classify the user as present.

Delegating Hotplug

Many USB devices serve a single purpose – when a user attaches a scanner, you can safely assume they want to scan something. If they attach a digital camera, they will typically want to transfer the images stored on the camera to the PC. This causality makes it possible to automate certain processes. For example, if you are interested in automating the process of transferring data between your home and work PCs, you can save yourself all that typing and automate the transfer.

The issue at stake is one of making hotplug events accessible to users. To allow this to happen, we need to extend the USB hotplug agent, and users need to write hotplug scripts for the devices they intend to support. Listing 6 shows the extension, which needs to be stored as */etc/hotplug/usb.specialdev* by the root users. To make sure the USB agent honors the extension, the root user also needs to add a *usb.specialdev* line to the *usb.agent* in the same directory. This has to be the last instruction in the *add* block right at the end of the file to ensure that the agent will process the

Listing 4: Net Link Agent

```
01 #!/bin/sh
02 (...)
03 case $ACTION in
04 add|register)
05 (...)
06   if grep -q
   "^allow-hotplug[[:
   space:]].*${INTERFACE}" /etc/
   network/interfaces; then
07     # this $INTERFACE is marked
   as class hotplug
08     if ps -C ifup ho args |
   grep -q "$INTERFACE"; then
09       debug_mesg "Already
   ifup-ing that interface"
10     else
11       start-stop-daemon
   --start --background --pidfile
   /var/run/hotplug.net.ifup.
   bogus \
12       --startas /etc/hotplug/
   net.ifup -- "$INTERFACE"
13     fi
14     exit 0
15 (...)
16   ;;
17 remove|unregister)
18 (...)
19   debug_mesg "Invoking ifdown
   $INTERFACE"
20   if [ -e /var/run/
   netplugd.${INTERFACE} ]; then
21     kill `cat /var/run/
   netplugd.${INTERFACE}`
22   fi
23   ifdown "${INTERFACE}"
24   ifconfig ${INTERFACE}
   0.0.0.0 up
25   start-stop-daemon --start
   --background --pidfile /var/
   run/netplugd.${INTERFACE} \
26   --exec /sbin/netplugd -- -i
   ${INTERFACE} -p /var/run/
   netplug
27 (...)
28   ;;
29 (...)
30 esac
```

Listing 5: bluenear

```
01 #!/bin/bash
02 export INTERFACE="00:01:E3:45:FF:FF"
03 STATE="away"
04
05 while true; do
06   hcitool cc ${INTERFACE} 2>/dev/null
07   if hcitool con|grep -q ${INTERFACE}; then
08     Signal="`hcitool lq ${INTERFACE}`"
09     hcitool dc ${INTERFACE}
10
11     if [ "${Signal##*: }" -lt 128 ]; then
12       if [ "$STATE" != "away" ]; then
13         STATE="away"
14         export ACTION="unregister"
15         /sbin/hotplug user
16       fi
17     else
18       if [ "$STATE" != "near" ]; then
19         STATE="near"
20         export ACTION="register"
21         /sbin/hotplug user
22       fi
23     fi
24   else
25     if [ "$STATE" != "away" ]; then
26       STATE="away"
27       export ACTION="unregister"
28       /sbin/hotplug user
29     fi
30   fi
31   sleep 5m
32 done
```

user-defined hotplug scripts after loading and initializing the drivers.

Lines 1 through 3 ensure that entities registered by hotplug really are devices and not just device features. It is easy to distinguish between the two: only genuine USB devices have vendor and product IDs. Lines 5 through 7 read this information and the serial numbers.

The loop starting in Line 9 processes */etc/passwd* line by line, locating the home directory for each user. If the script finds a user hotplug script in the *.hotplug* directory below the user's home directory, and if the name of the directory matches the vendor and product IDs of the device that was just plugged in (Line 13), Line 18 checks if the filename contains a serial number, and if this number matches the device's serial number.

If the user script is not restricted to a specific serial number, or if the numbers match, hotplug will call the user hotplug script in Line 22

with the privileges of the current user. It is important for *su* to use hotplug's environmental variables, that is, not to provide a login shell, as the environmental variables passed in by the kernel would otherwise be lost. Additionally, the script has to run in the background to remove the danger of hotplug blocking.

User Hotplug Scripts

Listing 7 shows a user hotplug script for a USB stick. Line 4 checks if this really is a USB storage device, and, if so, determines the device name. The script relies on a number of special features in order to perform its assigned tasks: hotplug uses the *DEVPATH* variable to return the path to the pseudo-files for the new device, although we do need to add a */sys* prefix here.

If the device is a USB storage device, a number of subdirectories will reside below this path. The subdirectories are named after the port position and host number.

If the directory at the bottom of the tree has a relative symbolic link for *block*, the device is a block device. *readlink -f* converts this relative symbolic link to an absolute path name, for example, */sys/block/sda*. And Line 9 takes the device name from this path name. Incidentally, there are more entries in */sys/block/sda*: *sda1*, for example, which represents the partition on a USB stick.

Users can add their own commands starting in Line 10, and hotplug will automatically run these commands with the privileges of the current user whenever the USB stick is plugged in. The most obvious options here would be to mount the partition and then call *rsync* or another tool to sync your data. In the case of a USB scanner, users would typically want to run *xsane*; or for a camera, they might like to transfer the images on the camera to the PC and sort the image files by date.

Users need to store scripts in the *.hotplug* subdirectory below their home directories and to ensure that the scripts are executable. The file name follows a pattern of *Vendor ID.Product ID:Serial number*, where the vendor and product IDs contain four digits and use small hexadecimal letters – for example, *0d7d.1600:075218833456*. The colon, and the serial number can be left out of the name if the script launches all devices with the vendor and product IDs in question.

Conclusion

The hotplug system is useful as it is, but you can do much more with it if you understand how it works. You can easily adapt the scripts described in this article for other purposes. Automatic data transfer, for instance, works for other device types as well as USB. A combination of Listings 6 and 7 with the Bluetooth device detection routine from Listing 5 would let hotplug autonomously update the MP3 files on any user's cellphones within range. You could also use this technique to sync calendar or address book data. There would be no need to attach these phones to the computer or launch a sync tool; in fact, the process would even work without the user logging in. ■

Listing 6: usb.specialdev

```
01 if [ ! -e "/sys${DEVPATH}/
   idProduct" -o ! -e "/
   sys${DEVPATH}/idVendor" ];
   then
02   exit 0
03 fi
04
05 read VendorID < /
   sys${DEVPATH}/idVendor
06 read ProductID < /
   sys${DEVPATH}/idProduct
07 read SerialNo < /
   sys${DEVPATH}/serial
08
09 for User in `cut -d":" -f1,6</
   etc/passwd`; do
10   UserName=${User%:*}
11   UserHome=${User#*:}
12
13   for HpScript in
   ${UserHome}/.hotplug/
   ${VendorID}.${ProductID}*; do
14     if [ -e "${HpScript}" ];
   then
15       UserSerial=${HpScript##*
   /}
16       UserSerial=${UserSerial/
   ${VendorID}.${ProductID}}
17       UserSerial=${UserSerial#
   :}
18       if [ -n "${UserSerial}"
   -a "${UserSerial}" !=
   "${SerialNo}" ]; then
19         break
20       fi
21       if [ -x $HpScript ]; then
22         su "$UserName" -c
   "$HpScript add" &
23       fi
24     fi
25   done
26 done
```

Listing 7: User Hotplug Script

```
01 #!/bin/bash
02 case $1 in
03 add)
04   BlockDev=`readlink -f /
   sys${DEVPATH}/*:*/
   host*/[0-9]*/block 2>/dev/
   null`
05   if [ -z "${BlockDev}" ];
   then
06     exit 1
07   fi
08
09   DevName=${BlockDev##*/}
10 (...)
```

INFO

[1] *netplugd*: *http://www.red-bean.com/~bos/*

[2] Scripts: *http://www.linux-magazine.com/Magazine/Downloads/71/hotplug*

THE AUTHOR

Mirko Dölle is the head of our Hardware Competence Center, and as such he tests more or less everything he can get his hands on – even if the lid is nailed down. In his leisure time, Mirko is the developer of the RoResc miniature rescue distribution, and the co-author of the LinVDR distribution. On the weekend, he makes the old alchemists' dream come real, turning gold into lead…