Proceedings of the Linux Symposium

Volume Two

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Conference Organizers

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

Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin

Dynamic Device Handling on the Modern Desktop

David Zeuthen
Red Hat, Inc.
davidz@redhat.com

Kay Sievers
Novell, Inc.
kay.sievers@suse.de

Abstract

Today, where almost all devices can be added and removed from a running system, the whole system environment needs to dynamically adopt to such changes. It creates the need to move from static device specific configurations given at install time to policy based runtime device configuration and change propagation troughout the whole system including currently running system services and end user applications.

Architectural Overview

The Linux kernel 2.6 exports almost all interesting internal device state in a special filesystem and sends out events for every change. Udev and HAL as system-wide services are picking up these device events, possibly request more information from the device itself or merge available information stored on the system. A global list of devices, including the whole device context is maintained and made available to every possible consumer. All changes to that list are propagated over a simple inter-process-communication (IPC) interface. Applications can subscribe to a specific class of changes and adapt itself according to that. Based on the notification the application has received, it can reflect that change accordingly or in response request a specific action to be taken for a specific device.

The Path of an Event

The core of the Linux kernel keeps the control over the interfaces where devices can be connected and disconnected at runtime. The kernel drivers create or destroy internal device instances, to handle and represent devices. These device instances are exported through the special filesystem *sysfs*. Most of the directories in sysfs represent a device and the files in the directories are properties or methods to request a specific action of the device:

```
/sys/class/block/sda
|-- dev
|-- device -> ../../devices/pci0000:00/\
        0000:00:1f.2/host0/target0:0:0/0:0:0:0
...
|   |-- read_ahead_kb
|   `-- scheduler
|-- range
|-- removable
|-- sda1
|   |-- dev
|   |-- size
|   |-- start
|   |-- stat
|   `-- uevent
`-- uevent
```

The hierachy of the device directories represent the dependency of the devices given by

the hardware itself and the logical stacking constructed by the kernel driver core. Every device is identified by its filesystem path, called the `devpath`. Everytime a device is added or removed an `uevent` is sent out over a kernel netlink socket:

```
add@/class/input/devices/input5
ACTION=add
DEVPATH=/class/input/devices/input5
SUBSYSTEM=input
SEQNUM=1166
...
```

A raw message send over the socket looks like this:

```
recv(3,"add@/class/input/devices/input5\0
ACTION=add\0
DEVPATH=/class/input/devices/input5\0
SUBSYSTEM=input\0SEQNUM=1166\0
CLASS=/class/input/devices\0
PHYSDEVPATH=/devices/pci0000:00/0000:00:1d.1/\
  usb2/2-2/2-2:1.0\0
PHYSDEVBUS=usb\0PHYSDEVDRIVER=usbhid\0
PRODUCT=3/46d/c03e/2000\0
NAME=\"Logitech USB-PS/2 Optical Mouse\"\0
PHYS=\"usb-0000:00:1d.1-2/input0\"\0
UNIQ=\"\"\0EV=7\0KEY=70000 0 0 0 0 0 0 0 0\0
REL=103\0", 2048, 0) = 383
```

An event sequence for a USB storage device looks like this:

```
add@/devices/pci0000:00/0000:00:1d.7/usb5/5-3
add@/devices/pci0000:00/0000:00:1d.7/\
  usb5/5-3/5-3:1.0
add@/class/scsi_host/host13
add@/class/usb_device/usbdev5.15
add@/block/sdb
add@/class/scsi_generic/sg1
add@/class/scsi_device/13:0:0:0

remove@/class/scsi_generic/sg1
remove@/class/scsi_device/12:0:0:0
remove@/block/sdb
remove@/class/scsi_host/host12
remove@/devices/pci0000:00/0000:00:1d.7/\
  usb5/5-3/5-3:1.0
remove@/class/usb_device/usbdev5.14
remove@/devices/pci0000:00/0000:00:1d.7/\
  usb5/5-3
```

The udev [1] daemon listens on the socket and matches the given event properties with a set of simple rules:

```
KERNEL=="mice",  NAME="input/$kernel_name"
ACTION=="add", MODALIAS=="?*", \
  RUN+="/sbin/modprobe $modalias"
SUBSYSTEM=="scsi_device", ACTION=="add", \
  RUN+="/sbin/modprobe sg"
KERNEL="sda[0-9]", \
  IMPORT{program}="/sbin/vol_id --export $tempnode"
ENV{ID_FS_UUID}=="?*", \
  SYMLINK+="disk/by-uuid/$env{ID_FS_UUID}"
RUN+="socket:/org/freedesktop/hal/udev_event"
```

Udev rules can specify the name of the device node to be created, request programs to be executed to import additional data into the event environment, or run external programs to setup or initialize a device, or request a matching module to be loaded into the kernel. Rules can also pass the whole event udev has received from the kernel, including possibly added data collected from the device or system configuration by executing a program or passing the event over a domain socket.

A simple database with persistent device information is maintained by udev and can be queried on demand by any program, but udev does not watch any device state besides adding or removal, and does not get notified about things like battery state changes, media changes in optical drives or card readers. The udev infrastructure is limited to very short living event handling to provide the initial setup of a device and to reflect the kernels internal device state to userspace.

All advanced subsystem specific knowledge and device monitoring requires a stateful system service that has specific knowlege about certain classes of hardware, and has access to information that can uniquely identify a device.

Abstraction and Meaningful Event Context

In order to provide applications with more information than what the kernel or udev can provide, all hardware information is kept in the

stateful HAL daemon. Upon startup, the daemon scans sysfs and builds a list of *device objects*. It connects to the udev daemon to get notified about further device state changes and updates its internal device representation from that on accordingly. Each device object is identified by a *Unique Device Identifier* (UDI) and each has a number of properties which simply are key/value pairs:

```
udi ='/org/freedesktop/Hal/devices/volume_uuid\
   _c66f3d19_2e10_44c0_9bd6_a8fefc476f7d'
 volume.partition.msdos_part_table_type = 130
 info.product = 'SWAP-hda2'
 volume.size = 1077511680
 volume.num_blocks = 2104515
 volume.block_size = 512
 volume.partition.number = 2
 info.capabilities = {'volume', 'block'}
 info.category = 'volume'
 volume.is_partition = true
 volume.is_disc = false
 volume.is_mounted_read_only = false
 volume.is_mounted = false
 volume.mount_point = ''
 volume.label = 'SWAP-hda2'
 volume.uuid = 'c66f3d19-2e10-44c0-\
   9bd6-a8fefc476f7d'
 volume.fsversion = '2'
 volume.fsusage = 'other'
 volume.fstype = 'swap'
 storage.model = ''
 block.storage_device = '/org/freedesktop/Hal/\
   devices/storage_serial_NP0JT48299J8'
 block.is_volume = true
 block.minor = 2
 block.major = 3
 block.device = '/dev/hda2'
```

These objects are exported via the `D-BUS` [2] system message bus and each object implements the `org.freedesktop.Hal.Device` interface with methods that applications can use among other things to query properties. On Linux, there is almost a one to one mapping between directories in sysfs and the device objects. When new devices are plugged in or out, device objects will appear and disappear and applications listening on the message bus can be notified. Depending on the device object, properties may change values over time, which also triggers notifications to subscribed applications.

Most of the properties of HAL device objects are derived from the contents of sysfs files, some are read from the actual hardware using ioctl's, some are merged from device information files and some are related to the actual device configuration. A device object has one or more *capabilities* and for each capability a number of properties must be present. Properties are name spaced; for example the capability `block` requires properties prefixed with `block`. Some properties are optional. All properties are strongly typed (integer, double, boolean, string, and list of strings are supported) and are defined in the HAL specification [3].

Once a device object is constructed it is matched by one or more *device information files* to merge additional properties. These files are simple XML files that define rules. For example:

```
<match key="storage.model"
  contains="Storage-CFC">
    <merge key="storage.drive_type"
      type="string">compact_flash</merge>
</match>
```

will match all device objects that contains the string `Storage-CFC` in the `storage.model` property and set the drive type to be a Compact Flash reader, while

```
<!-- Sony Ericsson Handys with Memory Stick (Duo) -->
<match key="@storage.physical_device:usb.vendor_id"
  int="0xfce">
    <!-- K750i -->
    <match key="@storage.physical_device:usb.product_id"
      int="0xd016">
        <merge key="storage.drive_type"
          type="string">memory_stick</merge>
    </match>
</match>
```

matches a specific USB device and sets the drive to be of type `memory stick`. Since the HAL specification [3] precisely defines the properties (including the range of values they can assume), an application can rely on this property to e.g. display the right icon for such drives that reads Compact Flash, MemoryStick and so forth.

To provide up to date information including when media is removed, HAL polls device

files. This is a necessary since e.g. the Linux kernel (rightly) refuses to do so, since it is not always necessary.

For each device object, HAL provides a sufficient number of properties for consumer applications to make intelligent choices about how to react when new devices are added or removed or when properties change. An example is the way HAL abstracts batteries:

```
udi = '/org/freedesktop/Hal/devices/acpi_BAT0'
  battery.charge_level.percentage = 99
  battery.charge_level.rate = 0
  battery.charge_level.last_full = 20020
  battery.charge_level.current = 19910
  battery.voltage.current = 12429
  battery.reporting.rate = 0  (0x0)
  battery.reporting.current = 19910
  battery.charge_level.capacity_state = 'ok'
  battery.rechargeable.is_discharging = false
  battery.rechargeable.is_charging = false
  battery.is_rechargeable = true
  battery.alarm.unit = 'mWh'
  battery.alarm.design = 1002
  battery.charge_level.unit = 'mWh'
  battery.charge_level.granularity_2 = 1
  battery.charge_level.granularity_1 = 1
  battery.charge_level.low = 200
  battery.charge_level.warning = 1002
  battery.charge_level.design = 47520
  battery.voltage.design = 10800
  battery.voltage.unit = 'mV'
  battery.technology = 'LION'
  battery.serial = '21805'
  battery.model = 'IBM-08K8193'
  battery.vendor = 'SANYO'
  battery.present = true
  battery.type = 'primary'
```

The `battery` capability is represented in the way that the information about batteries is independent of whether the information is collected via the ACPI or PMU power management subsystems. HAL also represent some uninterruptible power suppplies (UPS) devices connected via USB in exactly the same way. Also some wireless keyboards and mice hardware will have capability `battery` since HAL is able to report how much battery power is left.

Another kind of asynchronous information that HAL reports are button events. HAL defines the capability `button`:

```
udi = '/org/freedesktop/Hal/devices/acpi_PWRF'
  button.has_state = false
  button.type = 'power'

udi = '/org/freedesktop/Hal/devices/acpi_LID'
  button.has_state = true
  button.state.value = true
  button.type = 'lid'
```

and the above snippet shows the representation of the power button and the lid button. Since a button can be either pressed or remain pressed down, the abstraction has the notion of *state*. HAL creates device objects of capability `button` from several sources including ACPI (for power, sleep, lid buttons etc.), from connected keyboards (to catch auxillary buttons including sleep and eject buttons) and so forth. When HAL detects that a button is pressed, an asynchronous signal is emitted on the system message bus and applications can react accordingly. In the HAL universe this is known as a *DeviceCondition* and it carries some detail too.[1]

HAL supports the following capabilities:

- `volume`: Volumes
- `storage`: Drives
- `net`: Networking interfaces
- `input`: Input devices
- `printer`: Printers
- `portable_audio_player`: Portable music players (mainly flash drives)
- `alsa`: ALSA audio devices
- `oss`: OSS audio devices

[1] Notably, HAL also emits device conditions on HAL device objects representing optical drives when the eject button is pressed. This enables applications in the desktop to unmount and eject the disc and solves the well-known problem of nothing happening when the novice user presses eject on the drive. Sadly, some broken hardware does not supports this properly.

- `laptop_panel`: laptop panels
- `ac_adaptor`: AC adaptor
- `battery`: batteries
- `button`: special buttons on the system
- `camera`: for digital cameras (supported by e.g. gphoto2)

One example of a consumer application that relies solely on HAL for hardware information is the `gnome-power-manager` application [4]. In a nutshell, `gnome-power-manager` is the one-stop solution for system-wide power management on the GNOME desktop and it rivals and exceeds what properitary operating systems offers the end users. It includes neat things like user interface dialogs for configuring when to put the computer and display to sleep; it support UPS'es; display brightness, graphs of the discharge rate and so forth. In many ways it is a great showcase of how one can intelligent use the wealth of information stored in HAL.

Handling Events from the Consumers View

HAL was designed and architected primarily with desktop applications in mind. One of the goals was to make it easier to write desktop software to automate configuration of hardware with little or no user intervention. Another design goal was that end users should never ever have to edit configuration in `/etc` since end users are not expected to understand the file (they are all different formats) and they will not necessarily have privileges to do so.

Since UNIX-like operating systems are multi-user, it is necessary to store and read the device configuration settings from within the users session as users may configure hardware in different ways. On the GNOME desktop this means reading settings from the GNOME configuration system gconf [5].

Historically, software for configuring hardware wasn't designed with the desktop in mind. One reason for this is that configuring hardware requires super user privileges and until D-BUS emerged there was no good way of letting unprivileged applications perform privileged operations.

With HAL this has changed: each device object in HAL may export one or more capability specific interfaces to be invoked by software running in the user session. That way global device state change events travel from the system level through the user session, where the individual user policy is stored, back to the system to setup devices according to the users given preference.

For example device objects of capability *volume* exports the `org.freedesktop.Hal.Device.Volume` interface with the following methods:

- `Mount()`: for mounting a volume into the filesystem
- `Unmount()`: for unmounting a volume
- `Eject()`: for ejecting the volume

Each method may throw one or more expections, for example the `Mount()` method may throw the following exceptions (omitting prefix `org.freedesktop.Hal.Device.Volume`):

- `UnknownError`: Some unknown error occured

- `PermissionDenied`: The user is not allowed to mount the volume
- `AlreadyMounted`: The volume is already mounted
- `InvalidMountOption`: Attempted to mount with an invalid mount option
- `UnknownFilesystemType`: The file system is not supported on the system (missing file system driver for instance)
- `InvalidMountpoint`: The mount point specific is not allowed
- `MountPointNotAvailable`: The application requested a mount point that is not available
- `org.freedesktop.Hal.Device.PermissionDeniedByPolicy`: The caller lacked a *PolicyKit* privilege to carry out this operation. The name of the privilege is returned in the detail of the exception.

This format is a lot more predictable and error handling friendly than the traditional way of parsing the output of e.g. `mount(1)`. Other methods have well defined exceptions too.

At the time of writing, the following interfaces are supported by HAL (the list will have the `org.freedesktop.Hal.Device` prefix omitted):

- `Volume`: For mounting, unmounting, ejecting volumes / filesystems
- `Volume.Crypto`: For setting up LUKS volumes
- `SystemPowerManagement`: For suspend, hibernate, poweroff, reboot
- `LaptopPanel`: For display brightness on laptops; currently supports models from Toshiba, Asus, Panasonic, IBM/Lenovo, Sony, HP [2]

Since HAL uses D-BUS for IPC any unprivileged application may invoke device object methods and this needs to be limited a no to become a security issue. D-BUS by itself has a notion of security policy but this is, by definition, quite limited since D-BUS knows only little of the semantics of the object a given method is invoked on. To remedy this, HAL depends on PolicyKit [10] to check if the caller have privileges to invoke a given method on a given device object. This enables fine grained control that allows only some users to mount e.g. removable media while denying mounting of fixed hard disks. For more information see the PolicyKit documentation.

Current State and Future Plans

The combination of device properties, device conditions and capability specific methods provides an extensible and stable interface. At time of writing, HAL is already being ported to other operating system kernels and environments such as OpenSolaris [6]. All mainstream Linux distributions today ship udev and HAL and several desktop projects including GNOME (through Project Utopia [8]) and KDE (through Solid [9]) have HAL as a blessed [3] external dependency.

The task of "Making Hardware Just Work" is greater than the sum of *just* an OS kernel,

[2] This all relies on laptop specific ACPI modules as the kernel / X.org sadly lacks a standardized interface for this

[3] but optional since desktops normally supports all UNIX-like operating systems

drivers, udev, HAL, and the desktop environments; it requires integration and cooperation between developers of all projects.

One of the driving motivations for introducing HAL as a new layer between higher level software and the system device enumeration and discovery was to centralize the needed knowledge about specific device subsystems at a single location and provide a unified interface to that information. As an effect of this effort, a lot of higher level software ripped out its own device discovery and monitoring code and moved it into HAL. While at one side it is exactly what was intended, on the other side it increases the complexity in HAL and creates the need to get people involved with very specific subsystem knowledge at the low system level.

With the wide-spread use of HAL, which is the first generic system-wide device monitoring service, a lot of bugs in the Linux kernel were dicovered, cause likely no other software ever used the kernel interfaces that way or that extensively. A lot of these issues got fixed over time, but there is still a lot of improvement on the low-level side neccessary, to offer application developers a painless way to interact with the system.

HAL will continue to try to unify the view of the low-level interfaces to the higher-level applications. Its goal is to take over tasks that would require special defined policy in the kernel, to replace device and system specific knowledge from applications, and to replace or integrate current stand-alone system services into a unified interface for meaningful device information and classification to applications. Specific methods on device objects offered to applications will fully integrate the user session with the user owned preferences into the system-wide device handling.

Acknowledgements

Havoc Pennington's paper "Making Hardware Just Work" [7] kicked off the HAL project and it would never be a reality without help and support from Greg Kroah-Hartmann, Robert Love, Joe Shaw, Sjoerd Simons, the D-BUS developers, Red Hat, Novell, and many other people that thought about and worked closely toghether on the multiple layers of this software stack. We do not want to thank all the people that do nothing but complain about specific details they do not like, or complain about the fact that we need to change historical system behavior to be able to satisfy todays requirements, but at the same time use this software but never get their hands dirty or help moving forward and fixing all the problems we are facing.

References

[1] The Linux low-level Device Manager `http://www.kernel.org/pub/linux/utils/kernel/hotplug/udev.html`

[2] D-BUS is a message bus system, a simple way for applications to talk to one another. `http://www.freedesktop.org/wiki/Software/dbus`

[3] David Zeuthen, *et al.* HAL specification `http://webcvs.freedesktop.org/*checkout*/hal/hal/doc/spec/hal-spec.html`

[4] Richard Hughes. GNOME Power Manager `http://www.gnome.org/projects/gnome-power-manager/`

[5] Havoc Pennington, *et al.* GConf configuration system `http://www.gnome.org/projects/gconf/`

[6] Artem Kachitchkine, *et al.* Tamarack: Removable Media Enhancements in Solaris `http://opensolaris.org/os/project/tamarack/`

[7] Havoc Pennington. Making Hardware Just Work `http://ometer.com/hardware.html`

[8] Project Utopia Mailing List `http://mail.gnome.org/mailman/listinfo/utopia-list`

[9] Solid, The KDE Hardware Library `http://solid.kde.org/`

[10] David Zeuthen, *et al.* PolicyKit specification. `http://webcvs.freedesktop.org/*checkout*/hal/PolicyKit/doc/spec/polkit-spec.html`