Rules on how to access information in the Linux kernel sysfs

The kernel-exported sysfs exports internal kernel implementation details and depends on internal kernel structures and layout. It is agreed upon by the kernel developers that the Linux kernel does not provide a stable internal API. Therefore, there are aspects of the sysfs interface that may not be stable across kernel releases.

To minimize the risk of breaking users of sysfs, which are in most cases low-level userspace applications, with a new kernel release, the users of sysfs must follow some rules to use an as-abstract-as-possible way to access this filesystem. The current udev and HAL programs already implement this and users are encouraged to plug, if possible, into the abstractions these programs provide instead of accessing sysfs directly.

But if you really do want or need to access sysfs directly, please follow the following rules and then your programs should work with future versions of the sysfs interface.

- Do not use libsysfs
  It makes assumptions about sysfs which are not true. Its API does not offer any abstraction, it exposes all the kernel driver-core implementation details in its own API. Therefore it is not better than reading directories and opening the files yourself.
  Also, it is not actively maintained, in the sense of reflecting the current kernel development. The goal of providing a stable interface to sysfs has failed; it causes more problems than it solves. It violates many of the rules in this document.

- sysfs is always at /sys
  Parsing /proc/mounts is a waste of time. Other mount points are a system configuration bug you should not try to solve. For test cases, possibly support a SYSFS_PATH environment variable to overwrite the application's behavior, but never try to search for sysfs. Never try to mount it, if you are not an early boot script.

- devices are only "devices"
  There is no such thing like class-, bus-, physical devices, interfaces, and such that you can rely on in userspace. Everything is just simply a "device". Class-, bus-, physical, ... types are just kernel implementation details which should not be expected by applications that look for devices in sysfs.

  The properties of a device are:
    o devpath (/devices/pci0000:00/0000:00:1d.1/usb2/2-2/2-2:1.0)
      - identical to the DEVPATH value in the event sent from the kernel at device creation and removal
      - the unique key to the device at that point in time
      - the kernel's path to the device directory without the leading /sys, and always starting with with a slash
      - all elements of a devpath must be real directories. Symlinks pointing to /sys/devices must always be resolved to their real target and the target path must be used to access the device. That way the devpath to the device matches the devpath of the kernel used at event time.
      - using or exposing symlink values as elements in a devpath string

is a bug in the application

o kernel name (sda, tty, 0000:00:1f.2, ...)
- a directory name, identical to the last element of the devpath
- applications need to handle spaces and characters like '!' in the name

o subsystem (block, tty, pci, ...)
- simple string, never a path or a link
- retrieved by reading the "subsystem"-link and using only the last element of the target path

o driver (tg3, ata_piix, uhci_hcd)
- a simple string, which may contain spaces, never a path or a link
- it is retrieved by reading the "driver"-link and using only the last element of the target path
- devices which do not have "driver"-link just do not have a driver; copying the driver value in a child device context is a bug in the application

o attributes
- the files in the device directory or files below subdirectories of the same device directory
- accessing attributes reached by a symlink pointing to another device, like the "device"-link, is a bug in the application

Everything else is just a kernel driver-core implementation detail that should not be assumed to be stable across kernel releases.

- Properties of parent devices never belong into a child device.
Always look at the parent devices themselves for determining device context properties. If the device 'eth0' or 'sda' does not have a "driver"-link, then this device does not have a driver. Its value is empty. Never copy any property of the parent-device into a child-device. Parent device properties may change dynamically without any notice to the child device.

- Hierarchy in a single device tree
There is only one valid place in sysfs where hierarchy can be examined and this is below: /sys/devices.
It is planned that all device directories will end up in the tree below this directory.

- Classification by subsystem
There are currently three places for classification of devices: /sys/block, /sys/class and /sys/bus. It is planned that these will not contain any device directories themselves, but only flat lists of symlinks pointing to the unified /sys/devices tree.
All three places have completely different rules on how to access device information. It is planned to merge all three classification directories into one place at /sys/subsystem, following the layout of the bus directories. All buses and classes, including the converted block subsystem, will show up there.
The devices belonging to a subsystem will create a symlink in the

"devices" directory at /sys/subsystem/<name>/devices.

If /sys/subsystem exists, /sys/bus, /sys/class and /sys/block can be ignored. If it does not exist, you always have to scan all three places, as the kernel is free to move a subsystem from one place to the other, as long as the devices are still reachable by the same subsystem name.

Assuming /sys/class/<subsystem> and /sys/bus/<subsystem>, or /sys/block and /sys/class/block are not interchangeable is a bug in the application.

- Block
The converted block subsystem at /sys/class/block or /sys/subsystem/block will contain the links for disks and partitions at the same level, never in a hierarchy. Assuming the block subsystem to contain only disks and not partition devices in the same flat list is a bug in the application.

- "device"-link and <subsystem>:<kernel name>-links
Never depend on the "device"-link. The "device"-link is a workaround for the old layout, where class devices are not created in /sys/devices/ like the bus devices. If the link-resolving of a device directory does not end in /sys/devices/, you can use the "device"-link to find the parent devices in /sys/devices/. That is the single valid use of the "device"-link; it must never appear in any path as an element. Assuming the existence of the "device"-link for a device in /sys/devices/ is a bug in the application.
Accessing /sys/class/net/eth0/device is a bug in the application.

Never depend on the class-specific links back to the /sys/class directory. These links are also a workaround for the design mistake that class devices are not created in /sys/devices. If a device directory does not contain directories for child devices, these links may be used to find the child devices in /sys/class. That is the single valid use of these links; they must never appear in any path as an element. Assuming the existence of these links for devices which are real child device directories in the /sys/devices tree is a bug in the application.

It is planned to remove all these links when all class device directories live in /sys/devices.

- Position of devices along device chain can change.
Never depend on a specific parent device position in the devpath, or the chain of parent devices. The kernel is free to insert devices into the chain. You must always request the parent device you are looking for by its subsystem value. You need to walk up the chain until you find the device that matches the expected subsystem. Depending on a specific position of a parent device or exposing relative paths using "../" to access the chain of parents is a bug in the application.