The Linux Device File-System

Richard Gooch
EMC Corporation
rgooch@atnf.csiro.au

Abstract

The Device File-System (devfs) provides a powerful new device management mechanism for Linux. Unlike other existing and proposed device management schemes, it is powerful, flexible, scalable and efficient.

It is an alternative to conventional disc-based character and block special devices. Kernel device drivers can register devices by name rather than device numbers, and these device entries will appear in the file-system automatically.

Devfs provides an immediate benefit to system administrators, as it implements a device naming scheme which is more convenient for large systems (providing a topology-based name-space) and small systems (via a device-class based name-space) alike.

Device driver authors can benefit from devfs by avoiding the need to obtain an official device number or numbers, which are a diminishing resource, prior to releasing their product. Furthermore, they have complete autonomy in their assigned sub name-space, facilitating rapid product changes where necessary.

The full power of devfs is released when combined with devfsd, the devfs daemon. This combination allows administrators and developers to manage devices in new ways, from creating custom and virtual device name-spaces, to managing hot-plug devices (i.e. USB, PCMCIA and FireWire). Devfsd provides a lightweight, uniform mechanism for managing diverse device types.

1 Introduction

All Unix systems provide access to hardware via device drivers. These drivers need to provide entry points for user-space applications and system tools to access the hardware. Following the "everything is a file" philosophy of Unix, these entry points are exposed in the file name-space, and are called "device special files" or "device nodes".

This paper discusses how these device nodes are created and managed in conventional Unix systems and the limitations this scheme imposes. An alternative mechanism is then presented.

1.1 Device numbers

Conventional Unix systems have the concept of a "device number". Each instance of a driver and hardware component is assigned a unique device number. Within the kernel, this device number is used to refer to the hardware and driver instance. The device number is encoded in the appropriate device node, and the device nodes are stored on normal disc-based file-systems.

To aid management of device numbers, they are split into two components. These are called "major" and "minor" numbers. Each driver is assigned a major number. The minor number is used by the driver to determine which particular hardware instance is being accessed.

Prior to accessing a piece of hardware (for example, a disc drive), the appropriate device node(s) must be created. By convention, these are stored in the `/dev` directory.

1.1.1 Linux Implementation

In the Linux kernel, device numbers are currently stored in 16 bit integers. The major number component is allocated 8 bits, and the remaining 8 bits are used for the minor number. Each driver is thus allocated one of the 256 possible major numbers.

Each driver must register the major number it wishes to use, and the "driver operation methods" which must be called when file operations for that driver must be performed. These operations include opening, closing, reading, writing and others. The driver methods are stored in a table, indexed by major number, for later use. This table is called the "major table".

When a device node is opened (i.e. when a process executes the `open(2)` system call on that node), the major number is extracted and is used to index into the major table and determine the driver methods. These methods are then recorded into the "file structure", which is the in-kernel representation of the new file descriptor handle given to the process.

Subsequent operations on the file descriptor (such as reading and writing) will result in the driver methods being called, and thus control is transferred to the driver. The driver will then use the minor number to determine which hardware instance must be manipulated.

1.2 Limitations

Device numbers, traditional device nodes and the Linux implementation, have several limitations, discussed below.

1.2.1 Major and Minor size

Existing major and minor numbers are limited to 8 bits each. This is now a limiting factor for some drivers, particularly the SCSI disc driver, which originally consumed a single major number. Since 4 bits were assigned to the partition index (supporting 15 partitions per disc), this left 4 bits for the disc index. Thus, only 16 discs were supported.

A subsequent change reserved another 7 major numbers for SCSI discs, which has increased the number of supported discs to 128. While sufficient for medium enterprises, it is insufficient for large systems.

Large storage arrays can currently present thousands of logical volumes (one vendor can present 4096, and will soon double that figure). To support this under Linux would require the reservation of thousands of device numbers. This would rapidly erode the remaining device numbers, by consuming 16 or 32 major numbers for a single device driver. Combined with pressure from other device drivers, the device number space will soon be exhausted.

The limitation imposed by a 16 bit device number must be resolved if Linux is to grow in the enterprise market.

1.2.2 Device Number and Name Allocation

The conventional scheme requires the allocation of major and minor device numbers for each and every device. This means that a central co-ordinating authority is required to issue these device numbers (unless you're developing a "private" device driver), in order to preserve uniqueness.

In addition, the name of each device node must be co-ordinated by this central authority, so that applications will look for the same device names, and system administrators and distributors create the approved device nodes.

This system is not well suited to delegation of this responsibility. Thus, a bottleneck is introduced, which can delay the development and release of new devices and their drivers.

1.2.3 Information Duplication

Since device nodes are stored on a disc media, these must be created by the system administrator. For standard devices one can usually find a MAKEDEV programme which creates the thousands of device nodes in common use. Thus, for a change in any one of the hundreds of device drivers which requires a device name or number change, a corresponding change is required in the MAKEDEV programme, or else the system administrator creates device nodes by hand.

The fundamental problem is that there are multiple, separate databases of major and minor numbers and device names. Device numbers are stored in the following databases:

- inside the kernel and driver source code
- in the MAKEDEV programme
- in the `/dev` directory on many millions of computers

and device names are stored in the following databases:

- in the MAKEDEV programme
- in the `/dev` directory on many millions of computers
- in the source code of thousands of applications which access device nodes.

This is a classic case of information duplication, which results in "version skew", where one database is updated while another is not. This results in applications not finding or using the correct device nodes. There is no central database which, when changed, automatically changes all other databases.

1.2.4 `/dev` growth

A typical `/dev` has over 1200 nodes. Most of these devices simply don't exist because the hardware is not available on any one system. The reason for the large number of installed device nodes is to cater for the possibility of hardware that may be installed. A huge `/dev` increases the time to access devices, as directory and inode lookup operations will need to read more data from disc.

An example of how big `/dev` can grow is if we consider SCSI devices:

```
host          6  bits
channel       4  bits
id            4  bits
lun           3  bits
partition     6  bits
TOTAL         23 bits
```

This would require 8 Mega (1024*1024) inodes if all possible device nodes were stored. This would result in an impractically large `/dev` directory.

1.2.5 Node to driver methods translation

As discussed in section 1.1.1, each driver must store its driver methods in the major table. This table is 256 entries long, and is indexed upon each device open. If the size of device numbers were to be increased (in response the limitations discussed in section 1.2.1), then the major table would need to be converted to a list, since it would be impractical to store a table with a very large number of entries.

If the major table is converted to a list, this would require a list traversal for each device open. This is undesirable, as it would slow down device open operations. The effect could be reduced by using a hash function, but not eliminated.

1.2.6 `/dev` as a system administration tool

Because `/dev` must contain device nodes for all conceivable devices, it does not reflect the installed hardware on the system. Thus, it cannot serve as a system administration tool. There is no mechanism to determine all the installed hardware on a system.

It is possible to determine the presence of some hardware, as some device drivers report hardware through kernel messages or create informational files in `/proc`, but there is no uniform format, nor is it complete.

1.2.7 PTY security

Current pseudo-tty (pty) devices are owned by root and read-writable by everyone. The user of a pty-pair cannot change ownership/protections without having root privileges. Thus, programmes which allocate pseudo-tty devices and wish to set permissions on them must be privileged. This problem is caused by the storing of permissions on disc. Privileged programmes are a potential security risk, and should be avoided where possible.

This could be solved with a secure user-space daemon which runs as root and does the actual creation

of pty-pairs. Such a daemon would require modification to *every* programme that wants to use this new mechanism. It also slows down creation of pty-pairs.

2 The Alternative

The solution to these and other problems is to allow device drivers to create and manage their device nodes directly. To support this, a special device file-system (devfs) is implemented, and is mounted onto `/dev`. This file-system allows drivers to create, modify and remove entries.

This is not a new idea. Similar schemes have been implemented for FreeBSD, Solaris, BeOS, Plan 9, QNX and IRIX. The Linux implementation is more advanced, and includes a powerful device management daemon discussed in section 3.

2.1 Linux Implementation

The Linux implementation of devfs was initiated and developed by the author in January 1998, and was accepted for inclusion in the official kernel in February 2000. As well as implementing the core file-system itself, a large number of device drivers were modified to take advantage of this new file-system.

The Linux devfs provides an interface which allows device drivers to create, modify and destroy device nodes. When "registering" (creating) device nodes, the driver operation methods are provided by the driver, and these are recorded in the newly created entry. A generic pointer may also be provided by the driver, which may be subsequently used to identify a specific device instance.

Events in the file-system (initiated by drivers registering or unregistering entries, or user-space applications searching for or changing entries) may be selectively passed to a user-space device management daemon, discussed in section 3.

2.1.1 Naming Scheme

Each device driver, or class of device drivers, has been assigned a portion of the device name-space. In most cases, the names are different from the previous convention. The previous convention used a flat hierarchy, where all device nodes were kept directly in the `/dev` directory, rather than using sub-directories.

Devfs implements a hierarchical name-space which is designed to reflect the topology of the hardware as seen by the device drivers. This new naming scheme reduces the number of entries in the top-level `/dev` directory, which aids the administrator in navigating the available hardware.

Further, this scheme is better suited to automated tools which need to manipulate different types of devices, since all devices of the same type are found in a specific portion of the name-space, which does not contain device nodes for other device types. Thus, automated tools only need to know the name of the directory to process, rather than the names of all devices in that class.

2.2 Problems Solved

The limitations discussed in section 1.2 are revisited below, showing how devfs solves these problems.

2.2.1 Major and Minor size

Because the driver methods are stored in the device node itself, there is no need use device numbers to identify the driver and device instance. Thus, the limitations imposed by a 16 bit device number are completely avoided. An unlimited number of devices may be supported using this scheme.

2.2.2 Device Number and Name Allocation

By eliminating device numbers, there is no need for a central co-ordinating authority to allocate device numbers. There is still need for a central co-ordinate authority to allocate device names, but with the elimination of device numbers, this can be easily delegated. Each device driver can be allocated a

directory in the device name-space, by the central authority. The driver may then create device nodes under this directory, without fear of conflict with other drivers.

After allocation of the directory, the driver author is free to assign entries in that directory, without any reference to a central authority. If the driver author is also responsible for distributing the application(s) that must access these device nodes, then these applications can be changed at the same time as the driver is changed. Distributing the changed driver and application(s) at the same time, designers are free to re-engineer without fear of breaking installed systems.

2.2.3 Information Duplication

Since drivers now create their own device nodes, there is no need to maintain a MAKEDEV programme, nor is there a need to administer a `/dev` directory. This removes most cases of duplicated information. Device drivers are now the primary "database" of device information.

Thus, if a user installs a new version of a driver, there is no need to create a device node. This in turn means that there is no need for a device driver author to contact the maintainer of the MAKEDEV programme to update and release a new version of MAKEDEV. The user will not be called upon to down-load a new version of MAKEDEV, or manually create device nodes if a new version will not be available.

2.2.4 `/dev` growth

Since device drivers now register device nodes as hardware is detected, `/dev` no longer needs to be filled with thousands (and potentially millions) of device nodes that may be needed. Instead, `/dev` will only contain a small number of device nodes.

In addition, devfs eliminates all disc storage requirements for the `/dev` directory. An extfs inode consumes 128 bytes of media storage. With over a thousand device nodes, `/dev` can consume 128 kBytes or more. Reclaiming this space is of particular benefit to embedded systems, which have limited memory and storage resources. Installation floppy discs, with their small capacities, also benefit.

2.2.5 Node to driver methods translation

When drivers register device entries, their driver methods are recorded in the device node. This is later used when the device node is opened, eliminating the need to find the driver methods. Thus, there is a direct link between the driver and the device node.

This linking is architecturally better, as it eliminates a level of indirection, and thus improves code clarity as well as avoiding extra processing cycles.

2.2.6 `/dev` as a system administration tool

With `/dev` being managed by device drivers, it now also becomes an administrative tool. A simple listing of the directory will reveal which devices are currently available. For the first time, there is a way to determine the available devices. Furthermore, all devices are presented in a uniform way, as device nodes in devfs.

2.2.7 PTY security

Devfs allows a device driver to "tag" certain device files so that when an unopened device is opened, the ownerships are changed to the current effective uid and gid of the opening process, and the protections are changed to the default registered by the driver. When the device is closed ownership is set back to root and protections are set back to read-write for everybody.

This solves the problem of pseudo-terminal security, without the need to modify any programmes. The specialised "devpts" file-system has a similar feature for Unix98 pseudo-terminals, but this does not work for the classic Berkeley-style pseudo-terminals. The devfs implementation works for both pseudo-terminal variants.

2.3 Further Benefits

Besides overcoming the previously discussed limitations, devfs provides a number of other benefits, described below.

2.3.1 Read-only root file-system

Device nodes usually must reside on the root file-system, so that when mounting other file-systems, the device nodes for the corresponding disc media are available. Having device nodes on a read-only root file-system would prevent ownership and protection changes to these device nodes.

The most common need for changing permissions of device nodes is for terminal (tty) devices. Thus, it is impractical to mount a CD-ROM as the root file-system for a production system, since tty permissions will not be changeable. Similarly, the root file-system cannot reside on a ROM-FS (often used on embedded systems to save space).

A similar problem exists for systems where the root file-system is mounted from an NFS server. Multiple systems cannot mount the same NFS root file-system because there would be a conflict between the machines as device node permissions need to be changed.

These problems can be worked around by creating a RAMDISC at boot time, making an ext2 file-system in it, mounting it somewhere and copying the contents of `/dev` into it, then un-mounting it and mounting it over `/dev`.

I would argue that mounting devfs over `/dev` is a simpler solution, particularly given the other benefits that devfs provides.

2.3.2 Non-Unix root file-system

Non-Unix file-systems (such as NTFS) can't be used for a root file-system because they don't support device nodes. Having a separate disc-based or RAMDISC-based file-system mounted on `/dev` will not resolve this problem because device nodes are needed before these file-systems can be mounted.

Devfs can be mounted without any device nodes (because it is a virtual file-system), and thus avoids this problem.

An alternative solution is to use initrd to mount a RAMDISC initial root file-system (which is populated with a minimal set of device nodes), and then construct a new `/dev` in another RAMDISC, and finally switch to the non-Unix root file-system. This requires clever boot scripts and a fragile and conceptually complex boot procedure.

The approach of mounting devfs is more robust and conceptually simpler.

2.3.3 Intelligent device management

By providing a mechanism for device drivers to register device nodes, it is possible to send notifications to user-space when these registrations and unregistrations occur. This allows more sophisticated device management schemes and policies to be implemented.

Furthermore, a virtual file-system mounted onto `/dev` opens the possibility of capturing file-system events and notifying user-space. For example, opening a device node, or attempting to access a non-existent device node, can be used to trigger a specific action in user-space. This further enhances the level of sophistication possible in device management.

In section 3, the Linux devfs daemon is presented, which supports advanced device management.

2.3.4 Speculative Device Scanning

Consider an application (like cdparanoia) that needs to find all CD-ROM devices on the system (SCSI, IDE and other types), whether or not their respective modules are loaded. The application must speculatively open certain device nodes (such as `/dev/sr0` for the SCSI CD-ROMs) in order to make sure the module is loaded. If the module is not loaded, an attempt to open `/dev/sr0` will cause the driver to be automatically loaded.

This requires that all Linux distributions follow the standard device naming scheme. Some distributions chose to violate the standard and use other names (such as `/dev/scd0`). Devfs solves the naming problem, as the kernel presents a known set of names that applications may rely on.

The same application also needs to determine which devices are actually available on the system. With the existing system it needs to read the `/dev` directory and speculatively open each `/dev/sr*` device to determine if the device exists or not. With a large `/dev` this is an inefficient operation, especially

if there are many `/dev/sr*` nodes. In addition, each open operation may trigger the device to commence spinning the media, forcing the scanning operation to wait until the media is spinning at the rated speed (this can take several seconds per device).

With devfs, the application can open the `/dev/cdroms` directory (which triggers module auto-loading if required), and proceed to read `/dev/cdroms`. Since only available devices will have entries, there are no inefficiencies in directory scanning, and devices do not need to be speculatively opened to determine their existence. Furthermore, all types of CD-ROMs are presented in this directory, so the application does not have to be modified as new types of CD-ROMs are developed.

3 Advanced Device Management

Devfs implements a simple yet powerful protocol for communication with a device management daemon (`devfsd(8)`) which runs in user-space. It is possible to send a message (either synchronously or asynchronously) to `devfsd(8)` on any event, such as registration/un-registration of device entries, opening and closing devices, looking up inodes, scanning directories and more. This opens many possibilities for more advanced device management.

The daemon may be configured to take a variety of actions for any event type. These actions include setting permissions, running external programmes, loading modules, calling functions in shared objects, copying permissions to/from a database, and creating "compatibility" device entries. This yields enormous flexibility in the way devices are managed. Some of the more common ways these features are used include:

- device entry registration events can be used to change permissions of newly-created device nodes. This is one mechanism to control device permissions
- device entry registration events can be used to provide automatic mounting of file-systems when a new block device media is inserted into the drive
- device entry registration/un-registration events can be used to run programmes or scripts which perform further configuration operations on the devices. This is required for hot-plug devices which need to make complex policy decisions which cannot be made in kernel-space
- device entry registration/un-registration events can be used to create "compatibility" entries, so that applications which use the old-style device names will work without modification. This eases the transition from a non-devfs system to a devfs-only system
- asynchronous device open and close events can be used to implement clever permissions management. For example, the default permissions on `/dev/dsp` do not allow everybody to read from the device. This is sensible, as you don't want some remote user recording what you say at your console. However, the console user is also prevented from recording. This behaviour is not desirable. With asynchronous device open and close events, `devfsd(8)` can run a programme or script when console devices are opened to change the ownerships for *other* device nodes (such as `/dev/dsp`). On closure, a different script can be run to restore permissions
- synchronous device open events can be used to perform intelligent device access protections. Before the device driver open() method is called, the daemon must first validate the open attempt, by running an external programme or script. This is far more flexible than access control lists, as access can be determined on the basis of other system conditions instead of just the UID and GID.
- inode lookup events can be used to authenticate module auto-load requests. Instead of using kmod directly, the event is sent to `devfsd(8)`, which can implement arbitrary authentication before loading the module itself. For example, if the initiating process is owned by the console user, the module is loaded, otherwise it is not
- inode lookup events can also be used to construct arbitrary name-spaces, without having to resort to populating devfs with symlinks to devices that don't exist.

In addition to these applications, `devfsd(8)` may be used to manage devices in many other novel ways. This powerful daemon relies on two important features that devfs provides:

- a unified mechanism for drivers to publish device nodes
- a virtual file-system that can capture common VFS events.

See: `http://www.atnf.csiro.au/~rgooch/linux/` for more details.

4 Other Alternatives

Some of the limitations that devfs addresses have alternate proposed solutions. These do not solve all of the problems, but are described here for completeness, as are their respective limitations.

4.1 Why not just pass device create/remove events to a daemon?

Here the suggestion is to develop an API in the kernel so that devices can register create and remove events, and a daemon listens for those events. The daemon would then populate/de-populate `/dev` (which resides on disc).

This has several limitations:

- it only works for modules loaded and unloaded (or devices inserted and removed) after the kernel has finished booting. Without a database of events, there is no way the daemon could fully populate `/dev`
- if a database is added to this scheme, the question is then how to present that database to user-space. If it is a list of strings with embedded event codes which are passed through a pipe to the daemon, then this is only of use to the daemon. I argue that the natural way to present this data is via a file-system (since many of the events will be of a hierarchical nature), such as devfs. Presenting the data as a file-system makes it easy for the user to see what is available and also makes it easy to write scripts to scan the "database"
- the tight binding between device nodes and drivers is no longer possible (requiring the otherwise perfectly avoidable table lookups discussed in section 1.2.5)
- inode lookup events on `/dev` cannot be caught which in turn means that module auto-loading requires device nodes to be created. This is a problem, particularly for drivers where only a few inodes are created from a potentially large set
- this technique can't be used when the root FS is mounted read-only.

4.2 Just implement a better scsidev

This suggestion involves taking the `scsidev(8)` programme and extending it to scan for all devices, not just SCSI devices. The `scsidev(8)` programme works by scanning `/proc/scsi`.

This proposal has the following problems:

- this programme would need to be run every time a new module was loaded, which would slow down module loading and unloading
- the kernel does not currently provide a list of all devices available. Not all drivers register entries in `/proc` or generate kernel messages
- there is no uniform mechanism to register devices other than the devfs API
- implementing such an API is then the same as the proposal above in section 4.1

4.3 Put `/dev` on a ramdisc

This suggestion involves creating a ramdisc and populating it with device nodes and then mounting it over `/dev`.

Problems:

- this doesn't help when mounting the root file-system, since a device node is still required to do that
- if this technique is to be used for the root device node as well, initrd must be used. This complicates the booting sequence and makes it significantly harder to administer and configure. The initrd is essentially opaque, robbing the system administrator of easy configuration

- insufficient information is available to correctly populate the ramdisc. So we come back to the proposal in section 4.1 to "solve" this
- a ramdisc-based solution would take more kernel memory, since the backing store would be (at best) normal VFS inodes and dentries, which take 284 bytes and 112 bytes, respectively, for each entry. Compare that to 72 bytes for devfs

4.4 Do nothing: there's no problem

Some claims have been made that the existing scheme is fine. These claims ignore the following:

- device number size (8 bits each for major and minor) is a real limitation, and must be fixed somehow. Systems with large numbers of SCSI devices, for example, will continue to consume the remaining unallocated major numbers. Hot-plug busses such as USB will also need to push beyond the 8 bit minor limitation
- simply increasing the device number size is insufficient. Besides breaking many applications (no libc 5 application can handle larger device numbers), it doesn't solve the management issues of a `/dev` with thousands or more device nodes
- ignoring the problem of a huge `/dev` will not make it go away, and dismisses the legitimacy of a large number of people who want a dynamic `/dev`
- it does not address the problems of managing hot-plug devices
- the standard response then becomes: "write a device management daemon", which brings us back to the proposal of section 4.1.

5 Future Work

Devfs has been available and widely used since 1998. It has attracted a user-base numbering in the several thousands (possibly far greater), and forms a critical technology in SGI Pro-Pack (a modified version of Red Hat Linux) which is distributed by SGI for their Linux server products.

A number of improvements to devfs and to the generic kernel are needed to complete this work so that Linux will be suitable to the large enterprise and "data-centre" portions of the industry. These are discussed below.

5.1 Mounting via WWN

The current devfs name-space is a significant improvement on the old name-space, since the removal or addition of a SCSI disc does not affect the names of other SCSI discs. Thus, the system is more robust.

In larger systems, however, discs are often moved between different controllers (the interface between the computer and groups of discs). This is often done when a system is being reconfigured for the addition of more storage capacity. If discs are mounted using their locations, the administrator must manually update the configuration file which specifies the locations (usually `/etc/fstab`). Thus, some means of addressing the disc, irrespective of where it is located, is required.

Ideally, each device would have a unique identifier to facilitate tracking it. This unique identifier is defined by the SCSI 3 standard, and is term a WWN (world-wide number). If a disc is mounted by specifying its WWN, then it may be moved to a different controller without requiring further work by the administrator. This is important for a system with a large number of discs.

The SCSI sub-system in the Linux kernel needs to be modified to query devices for their WWNs, which can then be used to register a device entry which includes the WWN. All WWN entries would be placed in a single directory (such as `/dev/volumes/wwn` or `/dev/scsi/wwn`).

5.2 Mounting via volume label

For administrative reasons, some devices may be divided into a number of "logical volumes". This is often used for very large storage devices where different departments of an organisation are each given

a set of logical volumes for their private use. In this case, the storage may be presented as a single physical device, and thus would have a single WWN.

As with physical discs, logical volumes may need to be re-arranged for administrative reasons. Here, some mechanism which can address volumes by their contents is required. By storing a volume label on each volume, it is possible to address volumes by content.

Existing and planned logical volume managers need to be modified to support storing volume labels and must provide a common programming interface so that this information may be used in a generic way. Once these steps have been taken, volume labels may be exposed in the device name-space in a similar fashion as WWNs, placing entries in a directory such as `/dev/volumes/labels`.

5.3 Mounting via physical path

Prior to mounting via WWN or volume label, the initial location of a device is required. Once the device is located, the WWN may be obtained, or a volume label may be written. In order to initially locate the device, the physical path to the device must be used. To support this, device names which represent the physical location of devices are required. To support this, two new naming schemes are proposed.

A new `/dev/bus` hierarchy will be created, which reflects the logical enumeration of system busses, and sub-components thereof. For example, a specific PCI device (function2 in slot1 in PCI bus 0) would be represented by a directory named `/dev/bus/pci0/slot1/function2`. If this device was a SCSI controller, this directory would be the root of the SCSI host tree for this device. This naming scheme is a natural reflection of the Linux view of system busses.

Many larger systems have complicated topologies. For example, in a cc-NUMA system, I/O devices may be distributed across many nodes in the network. The detection order of busses may change if a new node is added, causing the existing identifiers in the `/dev/bus` heirarchy to change. A naming scheme is required which reflects the complete hardware topology. This is clearly vendor-specific, as different systems will have radically different topologies. Thus, designing a detailed structure to support different topologies is not feasible.

The solution I propose is to define a `/dev/hw` heirarchy, which is to be completely vendor-specific. This heirarchy will be created and managed by vendor-specific code, giving vendors complete flexibilty in their design. The `/dev/hw` heirarchy will effectively be a wiring diagram of the system. The only imposed standard is that the vendor remaps the generic Linux bus directories into the `dev/hw` tree. For example, `/dev/bus/pci0` would become a symbolic link to a directory somewhere in the `/dev/hw` tree.

The combination of these two naming schemes should provide sufficient flexibility for a wide variety of applications. The `/dev/bus` heirarchy will suffice for uncomplicated systems which do not change their topology (such as embedded and desktop machines, which dominate the market). In addition, `/dev/bus` provides a convenient place in which to search for all system busses, which is of use for the system administrator as well as some system management programmes. Also, because `/dev/bus` is managed by the generic Linux bus management subsystem, it is always available, even on systems with complex topologies. A vendor need not implement a `/dev/hw` heirarchy if it considers the benefits to be marginal, or if time does not permit prior to product shipment. Implementing a `/dev/hw` tree will add value, but is not required for basic operation of a system.

5.4 Block Device Layer

The Linux block device layer is used to access all types of random-access storage media (hard discs, CD-ROMs, DVD-ROMs and so on). This layer has two limitations. The first is that each block device is limited to 2 TB on 32 bit machines. This limits the maximum size storage device that can be attached to most Linux machines.

The second limitation is that the block device layer uses device numbers in all it's internal operations. These device numbers are used to identify devices and to lookup partition sizes and other configuration information. This limits the number of devices that can be attached to a Linux machine. While devfs *allows* device drivers to bypass device number limitations, the drivers must be changed to make

use of this. The block device layer is a critical layer that must be changed.

The block device layer (as well as the SCSI layer) needs to be modified to use device descriptor objects rather than device numbers. This is a significant, but essential, project. There has been some work on this already (a new `struct block_device` class has been defined), but more work is required.

5.5 Use of dcache for devfs internal database

The current implementation of devfs uses an internal database (a simple directory tree structure) to store device node entries. Much of the code used to manage this directory tree could be removed if the dcache (directory entry cache) in the VFS was used instead. This would significantly reduce the code size and complexity of devfs. The cost would be an increase in the memory consumption of devfs (from 72 to 112 bytes per device node entry).

When devfs was first implemented, this option was not available, but since then the VFS has matured, and this option should now be practical. A further modest change to the VFS is required (separation of dcache entries from VFS inodes).

This change is not required for large systems, as the existing implementation of the core file-system does not impose limitations. This change will provide a benefit for very small (embedded) systems, which have small numbers of devices, and thus the code savings outweigh the increased memory consumption in device node storage.

This change would also provide a political benefit, because of the code reduction, and would increase acceptance in some quarters. Despite its acceptance into the official kernel, devfs remains controversial, due its departure from traditional Unix device nodes.